Provided for non-commercial research and education use.
Not for reproduction, distribution or commercial use.

This article was published in an Elsevier journal. The attached copy is furnished to the author for non-commercial research and education use, including for instruction at the author's institution, sharing with colleagues and providing to institution administration.

Other uses, including reproduction and distribution, or selling or licensing copies, or posting to personal, institutional or third party websites are prohibited.

In most cases authors are permitted to post their version of the article (e.g. in Word or Tex form) to their personal website or institutional repository. Authors requiring further information regarding Elsevier's archiving and manuscript policies are encouraged to visit:

http://www.elsevier.com/copyright

Author's personal copy

Available online at www.sciencedirect.com
ScienceDirect
Int. J. Human-Computer Studies 66 (2008) 1–22
International Journal of Human-Computer Studies
www.elsevier.com/locate/ijhcs

Component-based tailorability: Enabling highly flexible software applications

Volker Wulf[a,b,*], Volkmar Pipek[a], Markus Won[c]

[a]*Institute for Information Systems, University of Siegen, Hölderlinstr. 3, 57068 Siegen, Germany*
[b]*Fraunhofer Institute for Applied Computer Science (FhG-FIT), Schloss Birlinghoven, 53754 Sankt Augustin, Germany*
[c]*International Institute for Socio-Informatics (IISI), Stiftsgasse 25, 53111 Bonn, Germany*

Received 6 October 2004; received in revised form 28 August 2007; accepted 29 August 2007
Communicated by M.B. Rosson
Available online 14 September 2007

Abstract

Component technologies are perceived as an important means to keep software architectures flexible. Flexibility offered by component technologies typically addresses software developers at design time. However, the design of software which should support social systems, such as work groups or communities, also demands 'use-time', or technically spoken, 'run-time' flexibility. In this paper, we summarize a decade of research efforts on component-based approaches to flexibilize groupware applications at run-time. We address the user as a 'casual programmer' who develops and individualizes software for his work context. To deal with the challenges of run-time flexibility, we developed a design approach which covers three levels: software architecture, user interface, and collaboration support. With regard to the software architecture, a component model, called FLEXIBEANS, has been developed. The FREEVOLVE platform serves as an environment in which component-based applications can be tailored at run-time. Additionally, we have developed three different types of graphical user interfaces, enabling users to tailor their applications by recomposing components. To enable collaborative tailoring activities, we have integrated functions that allow sharing component structures among users. We also present different types of support techniques which are integrated into the user interface in order to enable users' individual and collaborative tailoring activities. We conclude by elaborating on the notion of 'software infrastructure' which offers a holistic approach to support design activities of professional and non-professional programmers.

© 2007 Elsevier Ltd. All rights reserved.

Keywords: Tailorability; End user development; Component-based systems; CSCW

1. Introduction

The need for flexible software systems is well known and well addressed in the research areas of software engineering. Driven by the need to be more efficient in software development, the approaches worked towards a better reuse of code and an increased comprehensiveness of software architectures (e.g. object-oriented programming, component-based systems, and lately service-oriented architectures). In the research area of computer-supported cooperative work (CSCW), the driving force of flexibilization is a different one: software needs to be flexible in order to be adapted to new or changing work situations in its context of use. As a matter of fact, end users, not professional designers, typically take action to adapt their software applications according to the everyday routines and problems they encounter. Tools, techniques and methods that have been developed to make an evolutionary software engineering process more efficient, usually do not consider this 'end-user' aspect. Component-based technologies have gained considerable attention in this context (cf. e.g. Szyperski, 2002), since they offer 'black box' reusability (Ravichandran and Rothenberger, 2003) by making the integration of third-party components possible without having to know or manipulate their code.

*Corresponding author. Institute for Information Systems, University of Siegen, Hölderlinstr. 3, 57068 Siegen, Germany. Fax: +49 271 740 3384.
E-mail addresses: volker.wulf@uni-siegen.de (V. Wulf), volkmar.pipek@uni-siegen.de (V. Pipek), won@iisi.de (M. Won).

1071-5819/$ - see front matter © 2007 Elsevier Ltd. All rights reserved.
doi:10.1016/j.ijhcs.2007.08.007

Author's personal copy

Empirical studies confirmed that end users consider replacing software rather than re-programming it (e.g. Robertson, 1998). Component-based systems offer both ways of re-designing software, replacing as well as reprogramming, during use-time. Therefore, they can be seen as an interesting starting point to support end-user-oriented tailoring of software applications.

Flexibility of software artifacts has been a major research issue in human–computer interaction (HCI), from its beginnings. Since the individual abilities of specific users are diverse and develop constantly, suitability for individualization is an important principle for the design of the dialogue interface. In general, users were supposed to adapt the software artifacts according to their abilities and requirements (cf. Ackermann and Ulich, 1987; Fischer et al., 1987; Fischer and Girgensohn, 1990; ISO-9241, 1999). However, the scope of flexibility offered in early implementations was usually limited to simple parameterization of the dialogue interface. While this line of thought gave the users of software artifacts a more active role for the first time, it remained a problem to address change requests, requiring a deeper manipulation of the software artifact and higher levels of use-time flexibility.

Starting in the late 1980s, industrial demands, resulting from the diffusion of personal computers into organizations and the emergence of computer networks, led to research efforts to provide flexible information systems that offered a functionality for cooperation and collaboration modifiable by their users. While prior work aimed at the individual user, it was now a user group, an organization, or other social entities that needed flexibility in order to adopt computers for cooperative work (cf. Lieberman et al., 2006). Henderson and Kyng (1991) worked out the concept of tailorability to name these activities, and stressed the importance of being able to re-design and re-develop software during and/or in the context of use. Software artifacts and commercial products, as well as research prototypes with a tailorable functionality, have been developed. Regarding commercial products, spreadsheets and CAD systems were explored first. "Buttons" was one of the first highly tailorable research prototypes, where users could change the dialogue interface and functionality on different levels of complexity (MacLean et al., 1990). With the emergence of network applications, supporting collaborative activities such as communication, cooperation or knowledge exchange, the need for tailorable software artifacts increased (Schmidt, 1991; Bentley and Dourish, 1995; Wulf and Rohde, 1995). However, the distributed nature of these systems and the potential interdependencies of individual activities have posed additional challenges to the design of tailorable applications (cf. Oberquelle, 1993, 1994; Wulf et al., 1999; Stiemerling, 2000; Won et al., 2006).

In this paper, we summarize the results of a decade of research in flexibilizing groupware by means of component-based tailorability. We propose a conceptual framework to identify different levels of design challenges. Based on this framework, we present the different aspects of our work. Drawing on the notion of 'software infrastructure', we conclude by proposing design guidelines for tailorable applications.

2. A framework to study tailorability

Empirical as well as design-oriented research has indicated two major challenges in building tailorable systems (Mackay, 1990; MacLean et al., 1990; Nardi, 1993; Oppermann and Simm, 1994; Page et al., 1996; Wulf and Golombek, 2001a). The first challenge was to support re-design *during use*, and the second, to allow end users *within their use contexts* to take a leading role in re-designing their infrastructures. In particular, for the second challenge, important refinements have been described as:

- *Support for tailoring on different levels of complexity*: MacLean et al. (1990) have already pointed out the problem that a considerable improvement of users' skills is required when tailoring a software artifact goes beyond simple parameterization. They called this the 'customization gulf'. Beyond simple parameterization, profound system knowledge and programming skills would be required normally. Therefore, tailorable applications should offer a gentle slope of increasing the perceived complexity of tailoring architectures and interfaces to stimulate learning. Providing different levels of tailoring complexity could also tackle the problem of divergent skill levels among the users, a strategy or affordance for learning complex tasks that has been suggested earlier (Burton et al., 1984; Beringer, 2004).
- *Support for cooperative tailoring*: Empirical research indicates that tailoring activities are typically carried out collaboratively (Mackay, 1990; Nardi, 1993; Wulf and Golombek, 2001a). Users with less technical skills or motivation delegate part of the tailoring activity to local system administrators, power users[1] or gardeners that possess higher levels of technological skills, and benefit from receiving advice or reusing tailored artifacts created by more knowledgeable users.

We developed a framework to capture all relevant aspects (cf. Fig. 1), which will also guide our presentation of nearly a decade of research regarding these issues. In our perspective from the field of CSCW, tailorability is the characteristic of software offering end users manageable software flexibility at run-time/use-time. We distinguish three dimensions of addressing this challenge:

- *Architectural level*: The technological foundation of all efforts is the availability of an architectural flexibility that allows addressing different complexities of use-time

[1]Very experienced users who are not IT professionals (i.e. computer scientists) and who do not have any programming experience are here referred to as power users.

Author's personal copy

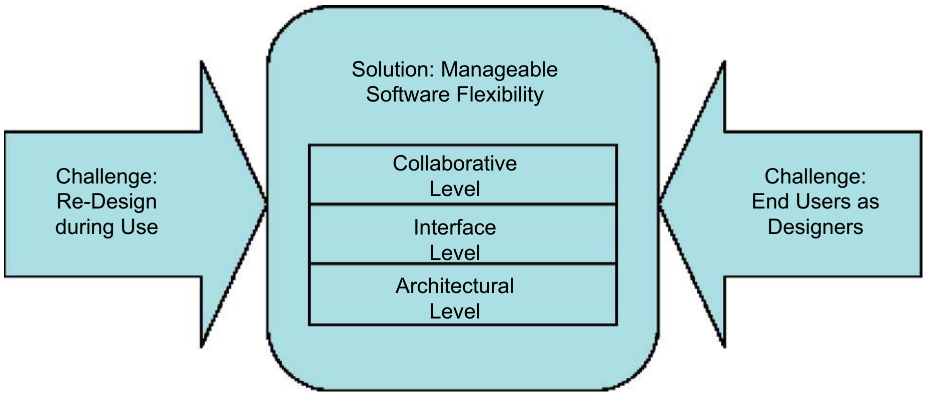

Solution: Manageable Software Flexibility
Collaborative Level
Interface Level
Architectural Level
Challenge: Re-Design during Use
Challenge: End Users as Designers

Fig. 1. End-user-oriented tailorability framework.

manipulations of the software artifact. While the maximum flexibility could be provided by using a publicly available source code, we are looking for a more pragmatic flexibility that respects the needs of the end users as 'casual programmers' and their work context at use-time (see Section 3).

- *Interface level*: End users as 'casual programmers' with very heterogeneous skills need specific support at the interface level. While simplifying programming tasks by providing appropriate interfaces has been addressed earlier, e.g. in research about 'Visual Programming', specific challenges result from the right abstraction level and granularity of concepts for an efficient re-design during use (cf. Section 4).
- *Collaborative level*: A community of non-professional programmers with potentially very diverse capabilities, motivations, and resources need technical support to share tailored artifacts. While different types of repositories support programmers to share source code and software modules, we assume that end users will have specific requirements.

Professional software designers focus on the implementation of a given specification, based on skills gained by education and training. In contrast, for a group of heterogeneously skilled end users who are designing-in-use, processes of understanding technologies, making sense of technologies, negotiating technology usage, and delegating and managing configuration work become a critical factor.

For every level, we discuss existing ideas against the two core challenges mentioned above, and suggest new approaches to develop and experiment with. We describe and connect experiences regarding a number of prototypes we have developed. All prototypes have been evaluated either in laboratory studies or in real use settings, but always with real end users from different organizations. About 80 end users from five organizations were involved in these studies, complemented with heuristic evaluation methods (Nielsen, 1993); in particular Thinking Aloud (Lewis, 1982) and Constructive Interaction (Kahler, 2000) as well as ethnographic methods (Blomberg et al., 1993) were applied. For those prototypes that we were able to evaluate in real world settings, we used questionnaires and semi-structured interviews to collect user feedback. For details on the evaluation of individual prototypes, we have to refer the reader to the publications dealing with the respective prototypes.

The purpose of this paper is to connect different research efforts in order to outline the dynamics and problems of providing tailorability for end users. While many of the individual research efforts have been published before, we now focus on the relations of this research on the levels described above, and on the remaining challenges. In general, we emphasize those aspects that do not have a strong correlation with classical software engineering approaches. We begin with describing the architectural level and the component-based tailoring platform FreEvolve, and then proceed to the interface level to describe end-user-oriented solutions to more specific problems. Next, we describe and discuss our experiences regarding cooperative tailoring, and finally, we will elaborate on overarching issues and further challenges.

3. Architectural level: component-based systems

To some extent, the discussion on component technologies in software engineering and the discussion on tailorable software artifacts in CSCW have a similar motivation: the differentiation and dynamics of the context in which software artifacts are applied. However, software engineering directs its attention towards the support of professional software developers during design time, while the concept of tailorability directs its attention towards users during run-time. In Section 1, we already made some points explaining why it is plausible to use component-based systems as a starting point for highly tailorable software applications. We now begin with describing our understanding of component-based systems, and then discuss flexibilization research from the field of CSCW to later attain some requirements for our work. In the final section of our coverage of the architectural level, we

Author's personal copy

describe how we overcame two specific problems by developing the FreEvolve platform as a basis for our work on component-based tailorability: the problem of component structures that are lost at run-time (use-time), and the problem of improving component intelligibility.

3.1. Basic concepts of component-based systems

The term "component" is not used very consistently within the software engineering community. We refer conceptually to Szyperski's (2002) notion of components. He gives the following definition:

> A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties (Szyperski, 2002, p. 41).

This idea of accelerating (re-)design by reusing modulized code was already discussed in the beginning of the software engineering discourse (McIlroy, 1968). Szyperski (2002) emphasizes the economic potentials of collaborative and distributed software engineering processes. Component technology allows applying the same software module in different artifacts (polymorphic composition). Modules need to be defined independently of each other. Their state can only be manipulated by interactions via well-defined interfaces. In this way, the developers who integrate an external component can access its services without understanding its implementation details. The visibility of a component's implementation may reach from black boxing (no visibility of the internal source code) to white boxing (full access to the internal code) with different levels of gray boxing (access to parts of the internal code).

This understanding of component technology has the potential to serve as a basis for the design of highly tailorable systems. Beyond parameterization and reprogramming, the composition of components provides a middle layer for the development of a gentle slope of tailoring complexity at an architectural level. Gray boxing also addresses the challenge of modeling nested component structures that provide additional potentials for differentiated complexity. Component-based architectures also help to address issues at other levels when dealing with flexibility: the strong encapsulation that the component concept provides can be helpful on the collaborative level, where end users should be supported in exchanging their tailored components. At the interface level, visualization concepts that aim at making architectural concepts understandable to end users may benefit from the diversity that gray boxing provides.

In software engineering research, there are many environments which allow modifying or composing components at design time. They usually generate applications that are monolithic after being compiled. For instance, in Visual Age for Java (IBM, 1998) or Visual Basic (Microsoft, 1996), the component metaphor is not perceivable at run-time unless it was anticipated and implemented by the designers.

However, there are component-based approaches which inspired our own work: the Darwin system (Magee et al., 1995), for example, is based on a component model that emphasizes the need for comprehensive gray boxing with a typed event-based interaction and hierarchical compositions. A process-oriented component enactment guarantees a certain level of flexibility during use. However, the system was intended to be used by trained administrators. Our motivation to address the needs of end users leads us to specifically focus on the comprehensiveness and the functional granularity provided by component architectures.

3.2. CSCW research on component-based tailorability

The area of CSCW includes research about concepts of and experiences with the development of groupware systems.[2] Here, we obtained our requirements for a component-based approach to tailoring. We now discuss a number of approaches for highly flexible groupware architectures that have been developed.

In the technology-oriented branch of CSCW research, tailorability is an important field of research. The Oval System (Malone et al., 1992) was one of the earliest approaches designed for use-time manipulation by end users. The main idea of Oval was to provide only four types of software modules (**O**bjects, **V**iews, **A**gents, and **L**inks) that can be used for building groupware applications. While the composition of these modules provides some functionality, it is not sufficiently fine-grained to allow building new applications without system-level programming. Prospero (Dourish, 1996) was an object-oriented framework that could be used to compose groupware applications. It offered a number of technological abstractions of functionality for collaboration (e.g. converging and diverging streams of cooperative work) developers could use to rapidly develop an application. Prospero addressed the concerns of developers of groupware systems, but it did not aim at enabling end users to tailor. DCWPL (Describing Collaborative Work Programming Language; Cortes, 1999) was a framework that allowed separating computational and coordination issues in the implementation of groupware applications. The computational modules were connected using coordinating modules that implemented the multi-user aspects of an application, and were described in a DCWPL file. Several language constructs could be used to describe session management, awareness support, etc. As DCWPL files were interpreted during run-time, tailoring was possible by changing this code. Like Prospero, DCWPL did not offer a tailoring environment directed at users with little programming experience. In the Tacts framework, Teege (2000) worked

[2]We denote the research field as CSCW while we use the term 'groupware' to refer to software applications in that field.

Author's personal copy

out the idea of 'feature-based composition' applied to tailor groupware. By adding a feature to a given software module, the functionality of an application could be changed during run-time. In his approach, the underlying architecture only provided two basic communication styles: a broadcast communication to all components and a direct connection between two components. The architecture remained open concerning the structure of the messages sent between components. As a result, all semantics had to be defined within the scope of the communicating components. Wang and Haake (2000) presented CHIPS, a hypermedia-based CSCW toolkit with elaborated abstraction concepts (role models, process models, cooperation modes, etc.) in a three-level modeling scheme (meta-model, model and instance), which allowed users to describe and tailor their cooperation scenarios. Generally, the system is based on an open hyperlink structure and is extendable on every modeling level in order to support every possible cooperation scenario. Wang and Haake (2000) focused on the notion of tailoring as a collaborative activity by using the meta-model to give the collaboration a reference framework to integrate it with the level of actual work.

In these systems and approaches, the authors have emphasized the following benefits:

- they implemented reusable and sharable structures,
- they preferred a building/construction metaphor over the 'text' metaphor of ordinary code,
- it was (with a varying intensity over the approaches) easy to group and re-group functionality at any time, and
- they defined 'building entities' as well as the data flows among them.

These considerations resemble what component-based systems offer at a deeper architectural level. While these approaches also render a component-based approach plausible, there are also some particularities that are not necessarily related to a component-based approach:

- the implementation of domain-oriented building blocks and representations (e.g. Oval; also Fischer and Girgensohn (1990) for a single-user application),
- the implementation of meta-models that guide tailoring (e.g. CHIPS), and
- a defined but flexible component communication (e.g. TACTS).

These aspects reveal that the authors of these concepts aimed at aligning structures at the 'interface layer' with the 'architectural' building block structures of the groupware applications. The successful implementation of these approaches and the encouraging impulses from the CSCW literature lead us to consider this as an important additional requirement for building highly tailorable systems.

In our research towards a component-based approach to tailorability, we aimed at combining concepts and experiences from both worlds: Software Engineering and CSCW. The concepts should provide a well-founded basis for re-use, encapsulation and composition, as well as implementational flexibility to provide end-user-oriented interfaces. There are obvious convergences in the ideas and approaches of flexibilization research in CSCW and of software engineering research on component-based systems. There are also new challenges to be addressed when using component technology to design tailorable groupware systems. By definition, tailoring is carried out after system installation and initialization by users who are not necessarily professional programmers. To allow composition after initialization or even during run-time, new concepts regarding the component model and the tailoring platform had to be developed. Since users are the key actors, appropriate tailoring interfaces and an application-oriented decomposition of the software functionality are needed. Moreover, technological mechanisms to support the sharing of tailored artifacts among the users need to be developed. In summary, we need

- to provide an architecture for re-designing software applications during use,
- to provide end-user-oriented concepts and interfaces, and
- to provide a strong congruency between architectural and interface concepts.

We developed the FreeEvolve architecture to address these requirements.

3.3. *FreeEvolve component platform and architecture*

To address the issues described above, we present results from research conducted at the University of Bonn and more recently at the University of Siegen. The design of tailorable groupware has been an important aspect of our work for almost a decade (e.g. Wulf, 1994, 2001; Stiemerling, 2000; Kahler, 2001a; Pipek, 2003; Won, 2003; Mørch et al., 2004; Wulf et al., 2005; Pipek and Kahler, 2006). Beyond component-based tailorability, we have also experimented with alternative approaches, such as rule-based architectures and the extension of off-the-shelf products. With regard to the latter, we will limit our presentation to those results applicable in the context of component-based approaches.

As a technical foundation, Won (1998), Hinken (1999), and Stiemerling (2000) have developed the FlexiBeans component model and the FreeEvolve tailoring platform.[3] Both were influenced by current technologies in software engineering, especially the JavaBeans component model and its run-time environment BeanBox.[4] However, due to the specific requirements of component-based tailorability

[3]In earlier versions, the FreeEvolve platform was called Evolve (cf. Stiemerling et al., 1999). The source code is available under GPL at: www.freevolve.org.

[4]The component model of JavaBeans and the source code of the BeanBox are publicly available from SUN and served as a base for early implementations.

Author's personal copy

for distributed systems, we had to refine the component model and to develop a distributed run-time and tailoring environment.

3.3.1. *Traceability and intelligibility of component systems*

Our first goal was to allow for a general intelligibility of component systems, more specifically, for an appropriate end-user-oriented traceability and visibility of current parameters within and of bindings between components. The work on the FlexiBeans model related to the requirement to achieve a high congruency between the architectural and the interface level.

In general, the atomic FlexiBeans components are implemented in Java, stored in binary format as Java class files, and packaged, if necessary, with other resources as JAR-files. We distinguish between the component and its instance. Every component can be instantiated in different compositions by different users at the same time. Each instance then has its own state. Like JavaBeans, the interaction between components is event-based. The state of an instance of a component can only change if the instance possesses the control flow, or through interaction with a component that is in possession of the control flow. The composition of the components determines which instances of components can interact (Stiemerling, 1998). In our approach, tailoring on the level of component composition happens by means of connectable ports. To allow tailorability at run-time, atomic and abstract (compound) components, as well as their event ports have to be visualized at the user interface.

The JavaBeans component model is based on typed events. Thus, events of the same type (e.g. button click event) are always received on the same port. Incoming events have to be analyzed in the receiving component, e.g. by parsing the event's source or by evaluating additional information that is sent together with the event. This approach makes it difficult for users to understand the different state transitions resulting from events of different sources (e.g. click events from two different buttons). An alternative strategy would be to use dynamically generated adapter objects in order to distinguish between different event sources (e.g. click buttons). Such an adapter object would forward different events of the same type (e.g. different button click events) to different handling methods depending on the event source. Both strategies hide part of the real components' interaction. An appropriate understanding of these strategies is essential for enabling the user to compose components appropriately. Therefore, the FlexiBeans component model has been developed to allow named ports, which are distinguishable according to types *and* names. Connections between components are only valid if the port's type *and* name match. Well-selected names of ports support an appropriate understanding of a component's semantics and its role within the entire assembly. Ports of the same name could have different polarity in the sense that they could either emit or receive a certain type of event.

3.3.2. *Composition techniques*

In traditional software engineering approaches, components are only visible at design time, and composing is done by means of a programming language or by a visual builder at design time. Since these approaches do not aim to allow any change of the component structure at run-time, it is lost after compilation. If the reconfiguration of components during run-time is supposed to be supported, the first challenge would be the design of not only a language that describes the composition of atomic components, but also the design of concepts to maintain these representations during run-time. As part of the FreEvolve concept, the CAT[5] component language was not only developed to deal with these issues, but also to describe compositions of atomic components into complex hierarchical structures. A CAT file, or in the distributed setting a set of CAT files (see Fig. 2), describes such a composition.

The CAT language supports hierarchically nested component structures to allow different levels of tailoring complexity. It is possible to build and store complex components by composing a set of atomic (or complex) components. From the user's point of view, there are two advantages: (a) the necessary number of abstract components to build the final application is lower than the number of atomic components would be and (b) their design can be more oriented towards specific application domains. Therefore, the necessary composition activities have a lower level of tailoring complexity than the composition activities at the atomic component level. The CAT language allows an arbitrary depth of nesting.

3.3.3. *Distributed tailoring platform*

Our focus in this contribution is to address the support of end-user interaction and ignore any software-technological difficulties we had to face during the development of our approach. These difficulties, in particular with regard to the implementation of the distributed features of our concepts (including the implementation of the dynamic reconfiguration of component networks), are extensively covered by Stiemerling and Cremers (1998) and Stiemerling et al. (1999a, b). Here, we only give a brief introduction to our basic concepts. The FreEvolve platform allows tailoring distributed applications with a tailoring environment/component editor embedded in a client–server architecture (cf. Fig. 2).

At the start, the tailorable application is stored persistently on the server. The atomic components are stored in JAR-files, while the current component structure is in a set of CAT files and *remote bind* files (DCAT files).[6] The user management provides the appropriate CAT and

[5]The syntax of the CAT (Component Architecture for Tailoring) is described in Stiemerling (1997). It draws on concepts already developed in port-based configuration languages such as DARWIN (cf. Magee et al., 1995) and OLAN (cf. Bellissard et al., 1996).

[6]To allow tailoring of client and server independently, we use three CAT files to describe one application. Two CAT files are needed to describe respectively the client and the server configuration of the components. The third description file, called DCAT file, describes the remote interaction

Author's personal copy

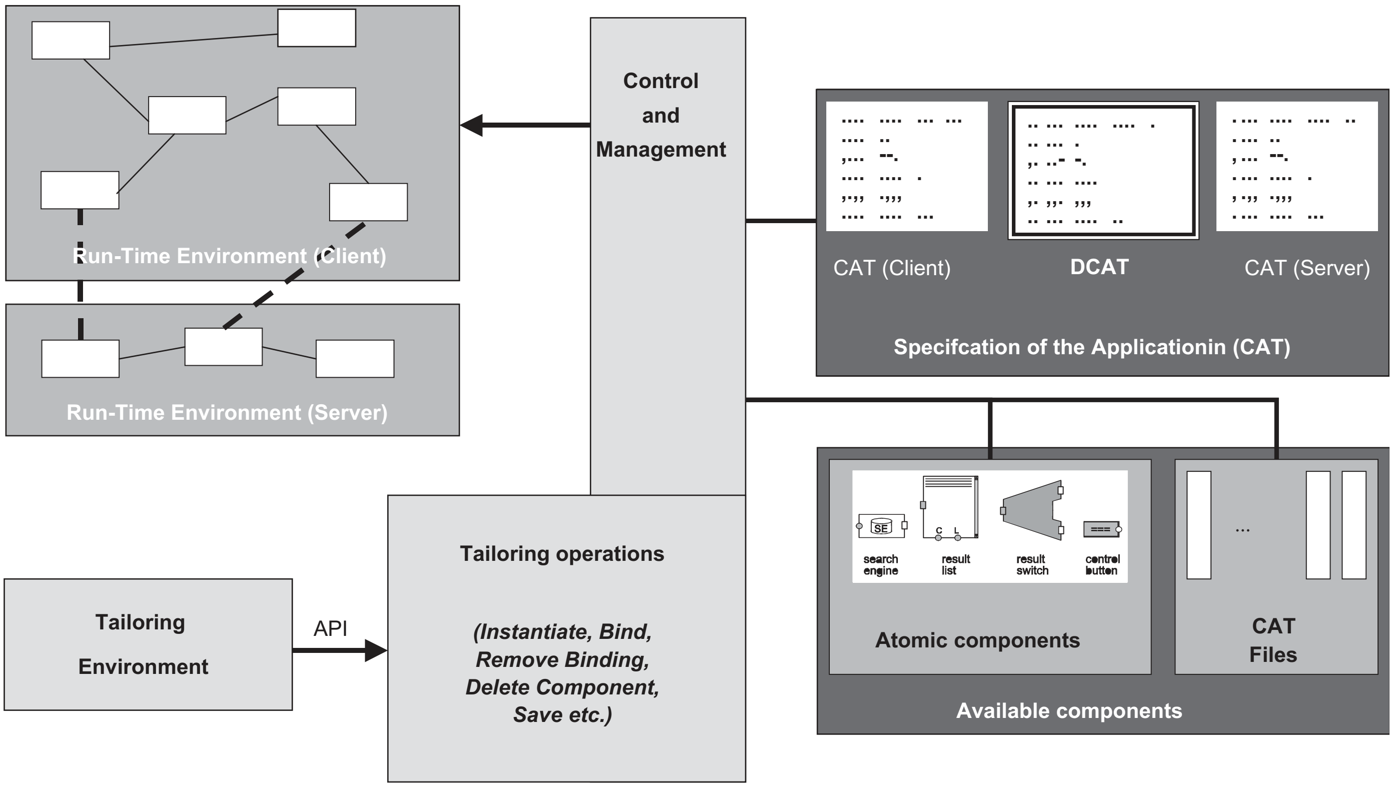


Fig. 2. FreEvolve platform.

DCAT files for each user who is starting her instance of the application. To support collaborative tailoring, every composition of the component can be shared with other users. The implementation does not allow run-time reconfiguration of the server-side composition.

During the start-up of an application, the CAT files on the server are being analyzed. All relevant components are represented at the server side. When user A logs in and starts the client of the tailorable application, the client connects to the FreEvolve server. The server then authenticates the user and sends the necessary components to the client. The client locally instantiates the atomic components and connects them according to the composition described in the associated CAT file. If the server application is not yet started, it will then be instantiated in the same way. Finally, client and server are connected according to the DCAT information.

The CAT files for the client's sides are stored on a central server that allows different users to run the same client by applying the same CAT file. Therefore, changes on the client side are transmitted to the server, stored persistently, and propagated to those active client machines that use the same client. With this type of distribution architecture, typical synchronization problems may occur and the propagation of tailoring activities to other users may lead to inconsistent system states in the FreEvolve platform. A specific protocol has been developed to recover a completely consistent version of the application in case of a breakdown (Stiemerling et al., 2000).

This architecture provides a well-organized structure of component-based CSCW applications even for a multi-user setting. It allows tailoring during the applications run-time, and it prepares for a higher level of congruency at the interface level.

4. Interface level: component visualization

There is a large body of work regarding the 'ergonomics' of programming, mainly in the field of Visual Programming (e.g. Shu, 1988; Myers, 1990; Ambler and Leopold, 1998; Pane et al., 2001). While many approaches aim to support the professional programmer (e.g. by aiming at visualizing the whole program and all its facets) there is also some work supporting less experienced programmers. The idea is to just visualize in an appropriate way what is necessary.

Repenning et al. (2000) constructed a simulation system where collaborative agents can be programmed by end users, offering a visual programming language which allows changes in the behavior of single agents. The interaction between the agents remains hidden and cannot be tailored, limiting the flexibility of possible applications. The approach did not aimed at providing a universal highly tailorable infrastructure, e.g. the programming language

(*footnote continued*)
between client and server components. Thus, every client application can be tailored without affecting the common server.

Author's personal copy

Visual AgenTalk needs to be re-designed for each field of application in order to be understood more easily.

The Regis system (Magee et al., 1995) allows the distribution of configuration management. Its two-dimensional (2D) environment implements many important techniques from Visual Programming. However, the underlying component model is based on the idea that a composition consists of only a few components, making it less scalable.

Some of these tools use multiple views to focus on different aspects of the application. For instance, in most programming environments, there is a code and a graphical user interface (GUI) view. In the field of end user tailorability, Mørch and Mehandjiev (2000) use this technique: their system ECHOES allows tailoring applications that can be seen and changed in different representation views.

In addition, other research inHCI aimed at supporting the learning of the functionality of single user applications. These concepts are based on either structuring, describing, experimenting or exemplifying the use of certain functions (e.g. Carroll and Carrithers, 1984; Carroll, 1987; Howes and Paynes, 1990; Yang, 1990; Paul, 1994). However, these approaches remain quite limited in the scope of their ability to communicate, a function's meaning because they usually just cover one specific aspect. Within the discussion on tailorability starting with Henderson and Kyng (1991), technology is not regarded to be the sole element of support for a user to understanding. On contrary, the complete socio-technical system, in which the user, the technology, and the other users are involved, is considered to be central.

A number of issues have been addressed in the HCI research described above: appropriate visualizing, flexibility, scalability, multiple views, and socio-technical concerns. In our research, we made use of this research (e.g. Visual Programming, as well as the socio-technical issues that contributed to our ideas at the collaborative level, which both will be discussed later), but in perceiving tailoring also as a user-driven process, we found the dynamic aspects of tailoring interaction of particular importance. It is not only the visual experience that is a key factor for allowing end users to familiarize themselves easily with the manipulation of IT artifacts. Very important is the way users are supported in the process of detecting and experimenting safely with the configuration options that allow them to learn about and grow into the role of a 'casual programmer'.

We have collected the experiences of our research as well as those of other researches dealing with the field of CSCW, and were able to refine four main challenges that cover different phases of the process of tailoring:

- *consistent anchoring*: the options to tailor a software artifact need to be indicated consistently;
- *intelligibility (of composition structures)*: the current composition structure of a tailorable software artifact (component aggregate) has to be represented intelligibly, especially in the dynamic aspects;
- *effect visualization*: the effects of tailoring activities have to be easily perceivable for the user; and
- *fault tolerance*: the tailoring environment should be fault tolerant in the sense that it indicates incorrect activities to the users and proposes advice.

We have developed several concepts and prototypes in dealing with these challenges. First of all, we begin with the concept of Direct Activation, which was developed to provide consistent anchors for tailoring functionality in component-based applications. Secondly, we describe issues of congruencies between different aspects and layers on which we can draw to provide users with an easy access to tailoring functionality. Thirdly, we suggest 'Exploration Environments' to increase the users' understanding of the dynamic aspects of the applications that they tailor. Finally, we address the issue of ''Fault Tolerance'' by suggesting additional constraint-based advisory structures to assure users that important dependencies within the component compositions will be guaranteed.

The work on the visualizations within the tailoring interface of FreEvolve was carried out by Won (1998, 2003), Hallenberger (2000), and Krüger (2003). Additional features to support tailoring activities were realized by Engelskirchen (2000), Golombek (2000), Wulf (2001), Krings (2002) and Won (2003).

4.1. *Easy access to tailoring functions: Direct Activation*

An empirical study of users of word processors indicated that finding the appropriate tailoring functions is a substantial barrier which either prevents tailoring, or adds significantly to its costs (Wulf and Golombek, 2001a). Discussing our findings in the context of earlier works (see Mackay, 1990; Page et al., 1996), we identified two rather distinct occasions where users want to tailor an application: (a) when a new version of the application is introduced and (b) when the users' current task requires a modified functionality. The users need different patterns of support in order to find tailoring functions in both of these situations.

For tailoring a new version of an application, providing a survey of the given tailoring functions would be appropriate to help tackle the finding problem. The users are informed about the scope of the new version's tailorability. When the users' current task requires a modified functionality, a context specific representation of the tailoring functions' access points would be appropriate. In such a situation, the user typically knows which aspects of the application he wants to modify, since the current version of the function hinders his work.

In order to tackle the second case, we have developed the concept of Direct Activation. Tailoring is needed when users perceive a state transition that does not lead to the intended effects following a function's execution. In this

Author's personal copy

case, users are typically aware of the function's anchor at the user interface. Therefore, the anchor of the tailoring function should be designed so that it 'relates' to the function to be tailored.

In our concept, three ways of 'relating' tailoring and its uses are distinguished. Firstly, by 'visual proximity to the use anchor', the visual representation of the anchor of the tailoring functions is placed closely to the anchor of the tailorable function. For example, in the case of certain parameters, the tailorable function has to be specified during activation, and visual proximity can be reached by displaying the anchor of the tailoring functions next to the one for specifying the parameters (e.g. in the same window). If the tailorable function is executed without any further specification from the menu or via an icon (a 'use anchor'), the anchor for the tailoring function will then be placed next to the one of the tailorable function. However, this does not work for what we call 'triggered functions' (functions that are not activated directly by the user, but initiated by some state changes in the application, e.g. email filters, cf. Oppermann and Simm, 1994). Thus, secondly, we suggest a 'visual proximity to the effect anchor' for those cases that do not provide a visible use anchor, although they cause perceivable effects at the interface.

A third, totally different approach within our framework, completely forgoes visualizations of anchors, instead it provides a 'consistent mode change' to the tailoring interfaces of functions. Mørch (1997) gives an example of a consistent mode to activate a tailoring function. In his system, a user can access different levels of tailoring functions by activating the function and pressing either the "option", or "shift", or "control" button. Restricted to specific functions, the Microsoft context menu supplies another example of how to design a 'consistent mode change' in order to activate tailoring functions. Whenever the display of a screen object may be tailored, a specific mouse operation allows accessing the tailoring function.

To evaluate the effectiveness of Direct Activation in finding tailoring functions, we have implemented prototypes and carried out an evaluation study. The results of this study show that Direct Activation eases tailoring activities (see Wulf and Golombek, 2001a).

4.2. *Visual tailoring environments: exploiting congruencies*

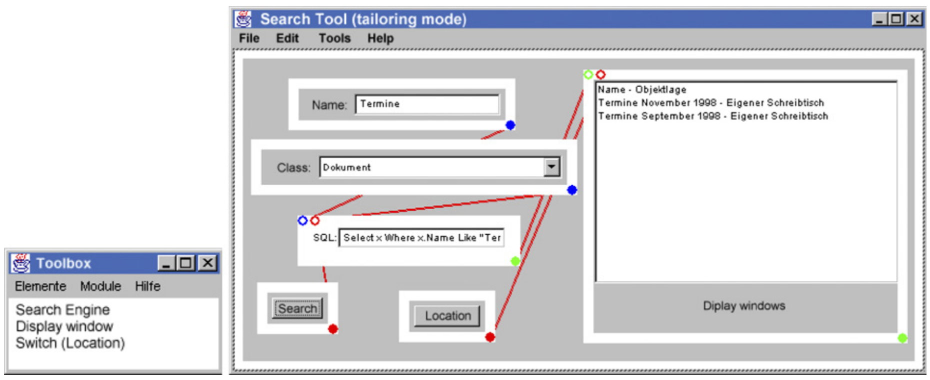
The general design of the tailoring interfaces in component-based systems aims at allowing 'natural' tailoring the way Pane and Myers (2006) postulated it. The goal is to create a visual tailoring environment where users are able to match between the interface at run-time and design time. Therefore, the users are supported to identify the properties that they want to tailor.

The components' characteristics and the composition metaphor inherent in the concept make a graphical/visual tailoring interface appear appropriate for changing the component structure. Within the graphical tailoring environment, the component structure of the software artifact is displayed as follows: components are visualized as boxes, ports are indicated as connectors at their surface, and the binding between two ports is represented by a line between these surface elements. Tailoring activities consist of adding or deleting (instances of) components and rewiring their interaction. During the course of our work, we have developed 2D (Stiemerling and Cremers, 1998; Won, 1998; Wulf, 2000) and 3D versions (Stiemerling et al., 2001) of the visual tailoring environment. In the following, we ignored the issue of parameterization of single components, which we have realized in all three of the tailoring environments by means of Direct Activation.

While we heavily made use of well-known ideas from Visual Programming, our main consideration was more directed at finding 'plausible congruencies' that would make it easier for end users to achieve their tailoring goals. We begin with an example (cf. Fig. 3).

The first tailorable application we developed was a search tool for a groupware application. The tailorable aspects were restricted to the client side, while the groupware application itself had a client–server architecture. Fig. 3 shows the 2D graphical environment to tailor the search tool.

The general approach is fairly straightforward: the users could compose variations of the search tool window, which consisted basically of different graphical elements in order to specify search queries and others in order to display search results. To allow these tailoring activities, the search



Name - Objektlage
Termine November 1998 - Eigener Schreibtisch
Termine September 1998 - Eigener Schreibtisch

Fig. 3. 2D graphical tailoring environment.

Author's personal copy

tool consisted of six types of atomic components. Four of these component types were visible during use (inquiry elements, start button, display elements) while the other two were visible only while tailoring (the search engine and the switches to direct search results towards specific graphical output elements). We made use of the categorizations of ports to represent them at the tailoring interface. The polarity of ports helped to distinguish between a component's input and output port: empty circles indicated input ports, filled circles indicated output ports. To support users in wiring the components appropriately, ports of the same type and name are given the same color, so that the users are directed to fit input and output ports by means of identical colors. Wired components were represented by using a connecting line between their corresponding ports. Abstract components were represented by a white frame around the atomic (or abstract) components they contained. If a power user has already designed several different abstract input and output components, other users can make their first steps in constructing their personal search tool by combining two abstract components.

Although the application resembles many other approaches to Visual Programming, it is important for us to clarify that we consciously aim to exploit 'plausible congruencies'. The first congruency is between the metaphor world of components and their visualization (Component-Metaphor Congruency). Component-based systems provide a specific perspective of the way software systems are built and how they can be manipulated. Attached to this perspective is a certain language (components, ports, connectors, etc.) and a set of related concepts. From our point of view, the art of providing tailoring interfaces is to maintain this congruency, while a certain domain-orientedness (e.g. by using appropriate naming schemes) is simultaneously provided. This allows users to understand tailoring options easily while still getting in touch with the concepts of the software architecture world they are working with.

The second congruency we exploited in order to support end users is the previously mentioned Architecture-Interface Congruency. It is important, from our perspective, that visualizations reflect the actual workings of the component architecture. Therefore, in the FlexiBeans concept, we did not only provide typed and directed, but also named ports to make the information and control flow among components as intelligible as possible. The above example also shows that 'gray boxing' is an important consideration. However, from this approach other problems, such as still having to differentiate the scope of tailoring activities referring either to the client or to the server side, remain unsolved.

The third important congruency is the Tailoring-Use Congruency. The idea is that the tailoring interface user should still be able to recognize the familiar users' interface with the corresponding visual anchors of functionalities. However, this proved to be quite difficult, and led us to conduct experiments with different two- and three-dimensional (2D and 3D) interfaces. The 2D approach presented so far has the advantage of enabling users to match directly between the run-time (use) environment and the tailoring environment. When a user changes into the tailoring mode, the visible components of the interface stay at the same place on the screen, while the invisible components, the ports and the connecting lines between them, are added to the display. The components that are invisible during run-time (search engine, switches for the results) are displayed at the same locations where they have been placed during the prior tailoring activities. However, a strict adherence to the third congruency has the consequence that screen locations, where invisible components are placed during tailoring, could not be used by visible components during use. This approach does not make efficient use of the screen space as long as larger parts of the invisible functionality stay "behind" the user interface.

To overcome these problems, a 3D graphical tailoring interface (cf. Fig. 4) has been developed. Spatial navigation ('flying through the model') can be used to explore applications. Components are represented as 3D boxes (with the components' names above them), which are located on a virtual plane. The ports are represented as rings around the components in order to facilitate connections from all directions. Like in the 2D case, the color indicates the type and name of the port. The polarity is expressed by the intensity of the color. The input port is represented by darker shading, while the output port is indicated by lighter shading. Connections between components are symbolized by tube-like objects linking the corresponding rings.

Like all other components, abstract components are represented by 3D boxes. However, the box's surface that encapsulates the containing components of lower hierarchical level becomes more transparent as the user navigates closer to it until the visual barrier completely disappears. Yet, the rings representing the ports of the abstract component remain visible. The user can now navigate or manipulate the inner-component structure. In our current implementation, atomic components remain black boxes even if the user navigates into their neighborhood. A gray-box strategy would allow navigation into an atomic component and inspection of the parts of the code that can be modified.

The distinction between client and server side is represented by a spatial arrangement which places the server in the center (represented by its tailorable component structure), while the different clients (represented by their tailorable component structure) are located in a semi-circle around the server.

In order to ease the transition from use into tailoring mode, we offer a reference between the visible components at the user's interface and the invisible components "behind" the screen. If a user changes into the tailoring mode, the actual client's GUI window will be projected into the 3D world. Light beams will then connect the

Author's personal copy

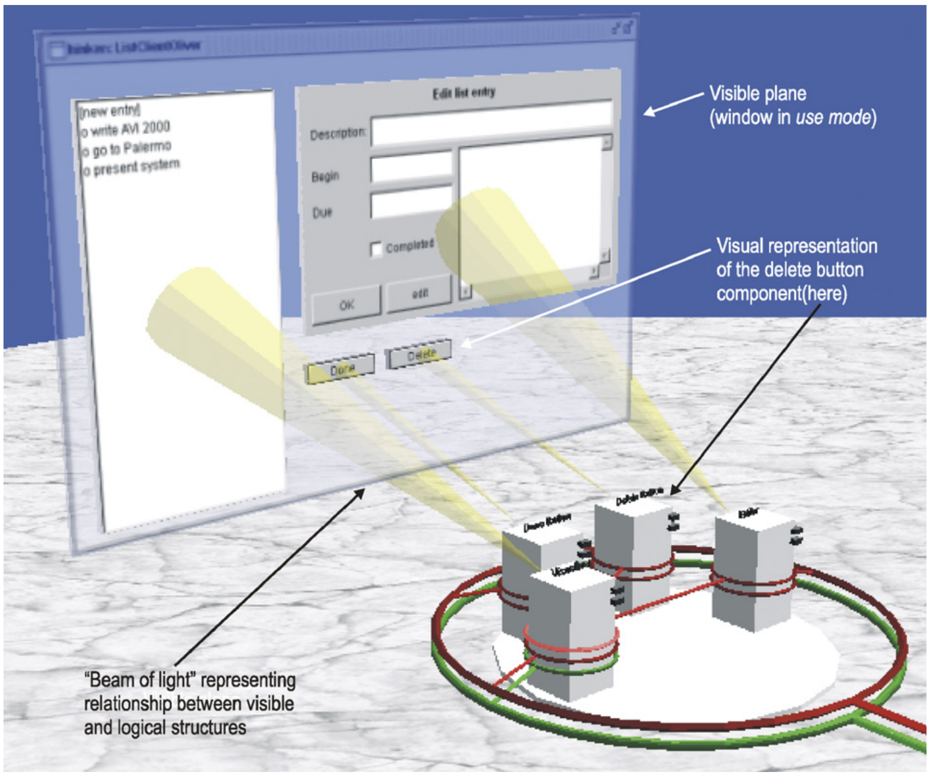


Fig. 4. 3D graphical tailoring interface displaying a client's tailorable component structure.

specific elements of the GUI interface to the client's component structure. When the user starts tailoring and enters the 3D world, he will be located in front of the GUI projection and will see the GUI on his regular interface. However, the semi-transparent plan also allows observing the component structure on the highest level of abstraction. Following the beams of light, he can navigate through the plane into the 3D space and explore or change the component structure.

With the development of the 2D and 3D solutions, it becomes obvious that the congruencies we want to establish may conflict with one another. In our implementations, it also becomes understandable that the Architecture-Interface Congruency is more effective at supporting the 3D environment, while the Tailoring-Use Congruency is better suited in the 2D environment. In user studies, we identified several problems with the 3D tailoring environment. Users had difficulties to navigate. Even after several trials, the time for reaching certain points in the architecture was considered too long. Moreover, the spatial arrangements led to situations where users might feel unable to navigate, because all relevant information (e.g. including the representation of the use mode) is visible on the screen. Our empirical evaluations also indicate that users have more problems in understanding the functionality and the use of invisible components (more abstract to them) than those of the visible ones (this applied to the 2D and the 3D interfaces).

The latter result leads to the development of a hybrid solution between 2D and 3D, which aims at addressing both congruencies separately. In the third prototype, we use separate windows to split the tailoring mode into three 2D vizualizations. One window maps the Tailoring-Use Congruency by visualizing only visible components (and making only those available for tailoring; cf. Fig. 5), whereas the other two map the Architecture-Interface Congruency by providing a 2D visualization of the component bindings on the client side (cf. Fig. 6, Composition Editor), and a tree view of the complete hierarchy of components (including server components; cf. Fig. 7, Component Explorer). All three windows are synchronized in order to make sure that they all represent the current state of composition.

For easier orientation and navigation, the synchronization feature also covers the highlighting and marking of components. Highlighting a component in one view leads to the component being highlighted in the other views simultaneously. The distinction of different views encourages users to use them for different tailoring tasks: resizing and/or re-arranging the components at the interface level is done in the editor for the visible components, and architectural modifications are done with the use of other editors. Changes of component parameters are possible in all three editors via context menus. In this approach, the notion of access control in component modification can also be applied to be able to deny access to the server-side components within the Component Explorer.

So far, we have only carried out a very preliminary evaluation study (see Krüger, 2003), indicating that even the least experienced users are able to work with all three different views provided by this tailoring environment.

Author's personal copy

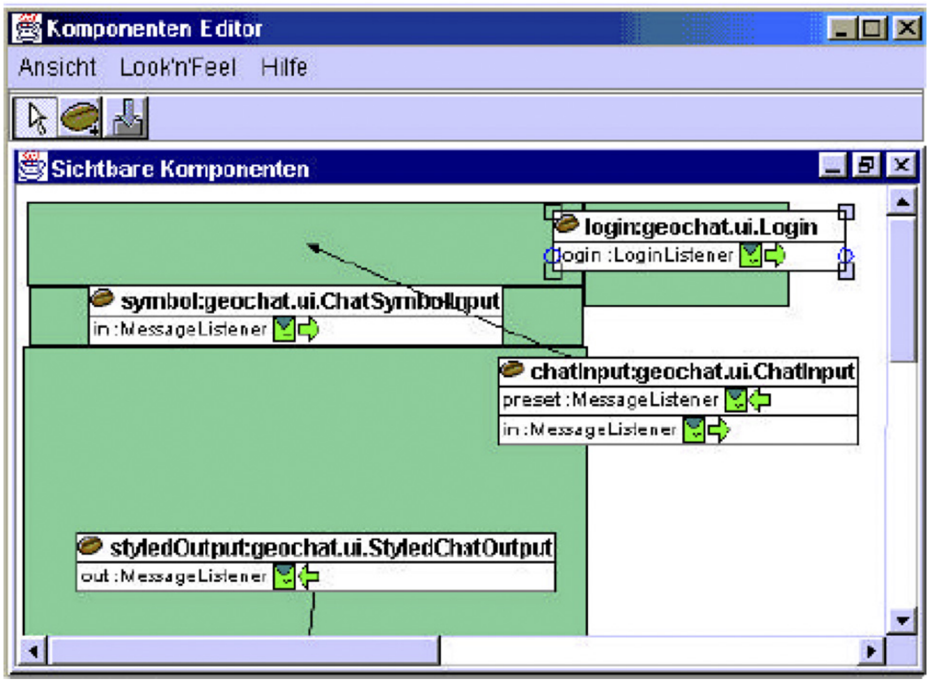


Fig. 5. View of visual components.

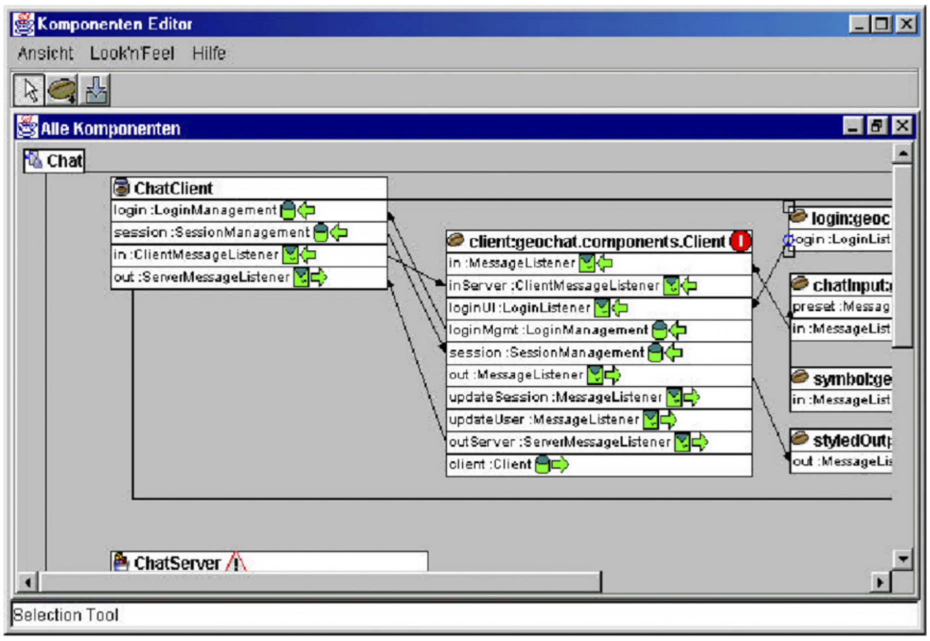
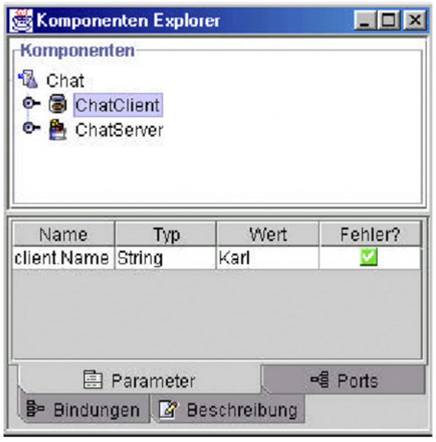

Komponenten Editor
Ansicht Look'n'Feel Hilfe
Alle Komponenten
Chat
ChatClient
login :LoginManagement
session :SessionManagement
in :ClientMessageListener
out :ServerMessageListener
client:geochat.components.Client
in :MessageListener
inServer :ClientMessageListener
loginUI :LoginListener
loginMgmt :LoginManagement
session :SessionManagement
out :MessageListener
updateSession :MessageListener
updateUser :MessageListener
outServer :ServerMessageListener
client :Client
login:geoc
login :LoginList
chatInput:
preset :Messag
in :MessageList
symbol:ge
in :MessageList
styledOut
out :MessageLis
ChatServer
Selection Tool

Fig. 6. View of the Composition Editor: end users can explore the network of sub-components, some details about ports and methods and their connection are displayed.

4.3. *Fault management: checking the integrity of component compositions*

Empirical studies indicate that the fear of destroying an application is one of the major obstacles in tailoring activities (Mackay, 1990). We have to consider the case that users may make mistakes when (re-)composing component structures. While these errors, on the one hand, threaten the functioning of the application (and even worse, the applications of others if the users work in a shared distributed environment), they are also regarded as opportunities for learning (cf. Frese et al., 1991). Although the differentiation of the ports (cf. FLEXIBEANS concept, typing, naming, directedness) already helps in preventing certain misconnections among components, technical mechanisms that actively detect errors in the composition of components (cf. Won, 2000, 2003) were additionally developed.

Author's personal copy



Name	Typ	Wert	Fehler?
client.Name	String	Karl	✓

Fig. 7. View of the Component Explorer: offers a different view on a component, providing more detail.

Such mechanisms for integrity checks are supposed to control the validity of the composition, to indicate the source of an error, give hints, and to correct the composition. The rule-based integrity check that we present here consists of two different concepts: (a) constraints and actions and (b) an analysis of the event flow.

Rule-based integrity checks are well-known in data base management systems (Silberschatz et al., 2001). Rules are terms in first order logic that can be evaluated automatically. This technique can be used to add external conditions to the use of components. By restricting the use of a component, these constraints describe the "right use" of them. For instance, if we have a set of GUI components, we can formulate a constraint like "all interface components have to have the same look and feel". If a user tailors an application (i.e. adds a new component) this condition can be checked.

The integrity constraints of all components are stored externally. Thus, they can be changed over time according to the users' or the organization's requirements. This separation may be useful in case standard components are added to the component framework. Parameters can be restricted with regard to the domain in which the component should be used. For instance, we introduce a watch component. This component is designed to work all over the world in order to support all time zones. Our domain in this case might be a regional organization set in Western Europe. Consequently, only one time zone is needed, and in this case, external constraints can ensure the time to be set to MET only.

Constraints include action descriptions. Action descriptions extend from providing simple text information, which may help the user to correct a wrong input, to describing an automated execution of solutions. Those action descriptions may cover all tailoring operations as discussed in our description of the architectural layer. Nevertheless, our goal here is not to compose applications automatically, but to ease the learning and understanding of tailoring activities. All the system interactions mentioned so far are represented at the user interface as they happen. As soon as a constraint fires, the corresponding component will be marked and the user may get detailed information by selecting it.

A second technical mechanism is the check of event flow integrity. As mentioned before, the FlexiBeans component model allows event-based component interaction, in which events are passed between independent components. In most cases, events that are created and passed to another component can be regarded as important information. Therefore, an event that is produced, but not consumed, may indicate a problem or an unsolved issue. Here is a short example, explaining such an integrity rule which might not be sufficient: Fig. 8(a) shows a three-component-composition consisting of a search engine (left), a filter component (middle) and an output window (right). If we mark consumers and producers in our constraint set, we can only ensure that the search engine and the output window are connected (in some way). However, we want to ensure that each created important event is also consumed in the right way. Thus, the event flow needs additional checking.

In order to implement the concept of event flow integrity, we classify the ports as either *essential* or *optional*: essential ports have to be connected to other components, whereas optional ports may not be used. For instance, the output component of the search tool, which displays the found objects' names, has two ports: one input port that is used for receiving search results from the search engine, and one output port that sends additional information (type, attributes) of selected search results to other components. In order to support users in building functional applications, the input port is classified as "essential", since it would be senseless to compose a search tool whose input window is not connected to any search engine. The output port here is classified as "optional", as one can use an output window without applying any additional information of the search results.

In order to deal with these dependencies, we have developed a 'regular providers/consumers' concept. For checking an event flow, all essential event ports have to be connected to 'regular providers' or 'regular consumers'. For instance, we may compose a search tool by connecting a search engine to a filter component (that filters some of the search results) connected to the output component. Here, the search engine has an essential output port. The filter component only passes events but it is not a consumer whereas the output component is a consumer of the search

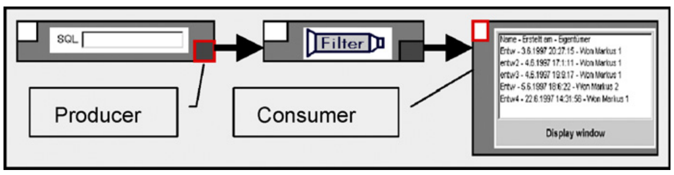


Fig. 8. Example of an Event flow.

Author's personal copy

result, and has an essential port that is finally connected. As a result, we have to differentiate the ports according to their function within a composition (producer, consumer, pass-though) and their need to be connected (essential, optional).

This information is described in external XML files which can only be changed by expert users. The integrity check is carried out by translating the composition into a Petri-net and then conducting an analysis (van der Aalst et al., 1997; Won, 2003). In case the event flow analysis detects an error, the user is provided with corresponding information within his tailoring environment. Components and their ports are marked if they are essential but not connected correctly.

4.4. *Spaces for learning: Exploration Environments*

To study tailoring empirically, Mackay (1990) as well as Oppermann and Simm (1994) stated the importance of explorative activities. We have developed the concept of Exploration Environments as an additional feature to support users in experimenting with tailorable groupware (Wulf, 2000; Wulf and Golombek, 2001b). An Exploration Environment allows simulating the execution of a tailorable groupware function by means of a specific system mode. This permits the learner's own user interface and the effects on other users' interfaces to be simulated on the output device. If a function is executed in the Exploration Environment, the effects of its execution on the simulated user interfaces will be similar to the effects the "real" function's execution has on the "real" user interfaces. In most groupware systems, users are only able to see parts of the system (their personal desktop, documents they are allowed to view, etc.). Exploration Environments have two main advantages: firstly, a user can interchange between different user interfaces. By executing a function in the Exploration Environment and switching between the simulations of her own and the other users' interfaces, a user can perceive how a newly tailored function works. Secondly, all data and documents of the Exploration Environment are accessible to the user. Thus, if a document cannot be found with a tailored version of a search tool, the user will know that he has to change the configuration of the search tool as he can see that a corresponding file exists in the Exploration Environment.

While research in HCI has already led to different exploration mechanisms (e.g. undo function, freezing point, or experimental data), the distributed character of groupware poses new challenges. Users are often unable to understand the way groupware functions work, since they cannot perceive the effects of the functions' execution at the other users' interface (e.g. the access rights granted to somebody else cannot be perceived by the owner). Only in cases where an application follows the WYSIWIS (What You See Is What I See) principle in a very strict manner (c.f. Johansen, 1988), users can perceive the effects of a function's execution on the interface of other users. In field studies (Wulf and Golombek, 2001b), we have learned that users try to overcome these problems by testing the results of their changes collaboratively. For instance, if users try to understand the current access right settings, they will ask their colleagues to try to open specific documents. This kind of exploration is often disturbing and difficult to do, in particular if the users are not collocated. To understand how tailorable functions are configured becomes even harder when their execution is not directly visible at the user interface. This type of changes in the configuration cannot be recognized directly. For instance, changes to an email filter can only be explored by sending a variety of different emails and tracking the recipient's mailbox.

We have built specific Exploration Environments for three different tailorable groupware tools: an awareness service, a search tool for groupware, and a highly flexible access control for a shared workspace. To evaluate the effectiveness of Exploration Environments in tailorable groupware, we have carried out a field study and an experiment in a lab setting. The results of these studies indicate that Exploration Environments support tailoring activities in groupware (cf. Wulf, 1999; Wulf and Golombek, 2001b). When designing Exploration Environments, we cannot remain in the realm of technology. To allow credible and realistic expeditions into the possibilities of alternative technology configurations, it is necessary to model at least parts of the organizational system, e.g. users, departments, roles.

5. Collaborative tailoring

Component-based software engineering is based on the assumption that software development can be organized best in a collaborative and distributed manner. Component repositories, together with monetary compensations for those who offer their source code for reuse, are supposed to render software development more efficient (see Szyperski, 2002). On the technical side, the issue is being addressed by providing a certain level of encapsulation of functionality, and by allowing an easy exchange of components. Empirical studies on tailoring activities of end users have also revealed a collaborative nature of tailoring (see Pipek and Kahler, 2006). While monetary compensation does not play an important role in these collaborations among users, different patterns of cooperative activities have been found among users who differ in their commitment to and qualification for tailoring.

Regarding tailoring, we deal with two ideal types of social design relationships: a relation between professional designers and users, and a relation among users who cooperate in the reconfiguration of the technology they use. In the FreEvolve approach, we anticipate that a certain division of labor would go with this distinction: on the level of the atomic components, the code for users is provided by professional developers, while on the level of the abstract components, users cooperate by providing each other with pre-integrated abstract components. Referring to the second type of relationship, we experimented with the design of

Author's personal copy

shared workspaces to exchange tailored artifacts among users. Moreover, additional features may add significantly to a component presentation for an intuitive reuse by others.

Engelskirchen (2000) and Wulf (2001) have developed a shared workspace to exchange abstract components within a groupware application: Höpfner (1998), Golombek (2000), and Kahler (2001a) have developed a shared repository to exchange tailored artifacts, such as document templates or button bars, among users of a word processor. Stevens (2002) has worked on metaphors that make visible and invisible components more comprehensive to users.

Repositories in software engineering are expected to contain components that could construct a wide variety of different applications. The necessities of navigation, the understanding of technological context, and choice of appropriate components are part of the professional knowledge of software engineers. Our experiences with end users (Wulf, 1999; Kahler, 2001b; Stevens and Wulf, 2002) revealed the need for a more application-oriented and user-oriented approach in designing repositories for the exchange of tailored artifacts. To allow a seamless transition between use and tailoring activities, the shared repository needs to be integrated into the tailoring environment and should be activated directly with only those components that are relevant to a certain tailoring context.

Repositories for tailored artifacts can be understood as shared workspaces where compositions of components can be exchanged. Shared workspaces have been discussed extensively in CSCW research (e.g. Bentley et al., 1997). In our case, however, shared workspace functionality needs to be integrated in both the groupware application and the additional features for making the components intelligible. Their basic functionality consists of uploading and downloading functions for (compositions of) components. Additional features allow specifying the visibility of compositions and defining access rights for different subgroups of users. Moreover, a notification service informs users as soon as relevant tailored artifacts are newly produced, modified, or applied. Such a service contributes to the mutual awareness of distributed tailoring activities, and may encourage the emergence of a tailoring culture (cf. Carter and Henderson, 1990). Since direct cooperation among users should also be supported, the tailoring environment may provide a function to mail tailored artifacts directly to specific users or user groups.

Summarizing, we have identified several design aspects for collaborative tailoring tools that support users in understanding the technology and its tailoring options as well as their own tailoring activities. Now, we will discuss the use of appropriate naming and classification schemes, look at the necessity of annotations, and finally address social aspects of Exploration Environments.

5.1. *Naming and classification schemes for components*

Users are typically confronted with a variety of different atomic and abstract components. By offering a number of atomic or abstract components, one has to provide meaningful names for them, since these names are listed at the interface. Due to limitations of screen space, the users' first choice is based on a very sparse visual presentation of the listed items. Naming the components in a meaningful way has turned out to be a key factor for the efficiency of tailoring activities.

Naming may still be straightforward regarding atomic components because they usually have only a limited function that is easier describable in a name. However, with abstract components in a shared repository, the functionality becomes more complex. Therefore, users have to agree on a shared vocabulary in describing tailored functionality. Some issues can be resolved by providing additional representations for invisible components, which explain their functionality (cf. Wulf, 1999). It can also be useful to work with a defined set of metaphors that is related to the application field (cf. Stevens et al., 2006).

A classification system is essential in order to structure a set of components. The classification of atomic components can benefit again from the simplicity of their purpose. For instance, in FreEvolve, we group them according to their ports (names, types, input/output) based on the assumption that their classification indicates the roles they can play within the composition (cf. Wulf, 1999). Nevertheless, for classifying abstract components, there are many plausible categories, e.g. technical aspects like the type of ports they use, or the organizational unit of the author of a component (used as an indicator for use/task similarities covered in a set of components). However, the most important aspect with regard to a naming and classification scheme is to recognize its potential to evolve. Therefore, it is important that names as well as classifications can emerge and develop as the users find and improve a common understanding.

5.2. *Annotation of components*

A field study indicated that users need additional support in distinguishing components beyond naming and classifying. Hence, we generated possibilities to textually describe atomic and abstract components. In order to describe atomic components, we have created a hypertext-based help menu that shows text to briefly explain the components' functionality and added screen shots when necessary.

Descriptions of abstract components have to be created by the users themselves. Therefore, we have implemented an annotation window that consists of the following text fields: "Name", "Creator," "Origin", "Description", and "Remarks". Since textual documentation of design rationales imposes an extra burden, it is often omitted (cf. Grudin, 1996). Thus, we tried to reduce the workload by providing automatic support in creating descriptions wherever possible (cf. Wulf, 1999). For instance, the "Creator" field is automatically generated by data taken from the user administration. The "Origin" field contains a reference

Author's personal copy

when an abstract component is created by modifying one already in existence. This reference is also automatically created. The "Description" field clarifies the behavior of an abstract component. If the component results from the modification of an existing one, the explanation of the original component is copied and put into italics, ready for editing. The "Creator" field may indicate on the one hand the quality of the abstract component. A composition becomes more trustworthy if it is created by someone who is an accepted expert within the community of users. On the other hand, the efforts become visible to the group, which leads to a higher degree of acceptance in many cases. Moreover, it may increase the motivation for sharing tailored artifacts as a service to a community of users.

5.3. *Experimenting with components*

Static descriptions already add a better understanding of the functionality that components offer. Still, it is a complex task to understand how a new component tailored by other users interacts with other known components. While the Exploration Environments described above support experimenting with completely assembled functionality, we have to find new approaches to support experimenting with atomic or abstract components. In case these modules do not cover an observable set of functionality, they cannot be executed in an Exploration Environment by themselves. Therefore, we have implemented an option which allows the users to store the "missing parts" together with the corresponding atomic or compound components. Together with the components themselves, the "missing parts" should provide a characteristic example for the component's use when building functionality. Those examples can then be executed in the Exploration Environment (cf. Wulf, 1999).

6. Discussion

We have described concepts and prototypes that were developed by our groups in order to provide tailoring means to end users of different work contexts. While many prototypes have been previously published in greater detail, our overview allows us to take a step back and have a look at the big picture.

The purposes of our research are twofold: we want to enable end users to independently tailor their technological infrastructure, and we want to allow them to do the tailoring at run-time. When we started with the research agenda, our choice of the component paradigm was motivated by its resemblance to tailoring concepts that were developed in the CSCW field. During the course of our work, we have developed more than 15 prototypes and have evaluated them with about 80 end users from 10 different organizations with various backgrounds, e.g. German federal government, steel industry, consultancy networks. The conceptual knowledge that was presented emerged from this research.

As a general outcome of our research, we can comment that the task of tailoring was more complex than we originally thought. From our point of view, it is also more complex than the task of (professional) programming. This is due to the fact that tailoring is not mainly about product development (which is often the guiding metaphor in software engineering), but 'infrastructuring'. Here, we take Star and Bowkers' (2002) notion, which defines an infrastructure as something that 'runs underneath' other structures, and is invisible, but becomes visible in the case of a breakdown. Infrastructures are used by people who are not experts in developing the infrastructure technologies. They familiarize themselves with the technological concepts driven by pure necessities of use. Users experience breakdown situations (be it an actual technological breakdown or simply a perceived incongruency between the expected and the delivered service of an application) which force them to learn new aspects necessary to understand and orchestrate technology. In this perspective, tailoring is much more than just a simple configuration of tools. It is an opportunity for reflection and learning about the (in-) abilities of today's technologies. 'Infrastructuring' tries to cover theses activities (Pipek and Syrjänen, 2006). In organizations, infrastructural problems usually are either addressed by a more or less appropriate division of labor (system administrators, IT departments, etc.) or left to the initiative of individual users. Approaches to 'infrastructuring' have to support the collaborations which result from this practice. We believe that software is a completely new type of infrastructural matter (compared to older infrastructures such as power lines, railroad systems or water pipes), since it *may* offer flexibility to support the activities of 'infrastructuring'. We should aim at allowing every user to dig deeper into technological matters at a level considered appropriate to him. Component-based tailorability allows realizing such a required 'gentle slope' of increasing complexity.

6.1. *Exploiting congruencies*

Star and Bowker (2002) explicitly referred to the historic dimensions of infrastructures, and described how new infrastructures emerge from older ones, and how concepts are being transferred between old and new infrastructures. They point out that for the individuals involved, historical aspects such as earlier experiences and existing expertise count. We described that in the form of 'congruencies' the architecture and the visualization of a component-based system should strive to maintain. Our choice of using a component-based approach was proved to be successful, since we were able to achieve these congruencies up to a certain level. The Component-Metaphor Congruency addressed the issue that the use of the component metaphor in programming is related to the notion of an, almost haptic, experience in building complex systems from simple building blocks that users may know from other construction domains. To achieve this congruency, it was

Author's personal copy

necessary to combine a strong encapsulation of all concepts that describe a component with clear visual indicators of how components fit and work together.

Related to the Architecture-Interface Congruency is a decision that underlies all our efforts: To provide a 'truthful' representation of the architecture at the interface, clear architectural concepts that can simply represent the user's interface (e.g. the port description we provided in the FlexiBeans concept) are needed. We accept that for a large number of individual problems, it would be easier for users to have an interface specifically designed for certain tailoring tasks. However, the research on the development of infrastructures leads us to believe that such a design could prevent users from building a deeper knowledge of the true complexity of the system, and may in fact hinder the understanding of future breakdown situations. We believe that for bridging the gap between component-based structures and the intuitive understanding of users in an application field, an additional metaphor framework on top of the component network could be the solution. But even in this case it should be possible to trace issues into deeper levels of the application's software architecture to further familiarize with the new infrastructure. Our experiences also confirm that it is not necessarily a good idea to maintain a strong separation between the computational and the interface level of applications (cf. Kuutti and Bannon, 1993).

The Use-Tailoring Congruency helps users to enter the world of tailoring and to understand specific tailoring options. In general, usage knowledge is less complex and acquired earlier than tailoring knowledge. The congruency helps the tailoring user to draw on earlier use experiences.

The three congruencies address the end user as a 'casual programmer', and allow him to interact with the infrastructure in a more powerful way.

6.2. *Employing a holistic approach*

We also like to emphasize here that the completeness of the research (in terms of addressing the architectural level, the interface level, and the cooperation level) was not our intention in the beginning, but it later proved to be essential during our evaluations. We simply cannot escape the embeddedness of users in their work context and their focus on actual tasks (where tailoring is more a side task). Therefore, we believe, scenarios that focus on providing help in just one specific situation do not cover the complexity of the problem. Apart from an incident-oriented support, we also have to address a strategic level of users having to familiarize with technologies. There, support on the individual as well as on the social level is needed.

So far, we maintain an understanding of tailoring as a task to reconfigure or redesign software tools. Henderson and Kyng (1991) already went beyond this understanding by employing a perspective, where tailoring is not only the design of tool configurations, but also the design of a tool's usages. There is usually a certain set of tasks, roles, use scenarios, and conventions associated with the technological reconfiguration of tools. Under the term of 'appropriation work', we address the activities around the making sense of technologies in an application field (Pipek, 2005). Beyond learning about the functionality of an application, it also deals with social activities related to the design of an appropriate usage of the application. An approach aware of the infrastructural character of software artifacts has to provide support for these activities. In the FreEvolve approach, we address that issue by combining component repositories with Exploration Environments. Pipek (2005) suggested the use of 'use discourse environments' to provide a platform for these negotiations. In general, it is an interesting goal to support users as a 'virtual community of technology practice'.

We want to close our discussion by addressing open ends in technology-oriented research on tailorability. While our results stem from long-term research activities covering the design and implementation of technological innovations as well as their evaluation in laboratory-settings and field studies, many issues still remain open. We do not know yet which type of tailorable applications is best suited for component-based approaches compared to other software technical paradigms such as rule-based or agent-based ones. From a technological perspective, our experiences indicate that applications or parts of them whose control flow can be presented in a rather linear order seem to be well suited for component-based tailorability.

Methods that allow finding appropriate modularizations of an application into atomic components are another issue for further investigation (cf. Stiemerling et al., 1997; Stiemerling, 2000; Stevens and Wulf, 2002). The modularization must also be meaningful to users (cf. Stevens et al., 2006). For different classes of applications, we need to find meaningful metaphors to communicate the meaning of individual components. The CoCoWare platform (Slagter et al., 2001) allows users to compose their own component-based groupware applications. In this platform, each component is in itself a small application, e.g. a session control or a conference manager component. From the software-technological perspective, their work is closely related to ours but with a main difference in the granularity of the components. While in CoCoWare components represent whole applications, FlexiBeans are conceptualized to be more fine-grained. On the one hand this allows more flexibility; on the other hand, tailoring becomes more complex.

In extending our approach, one can also imagine introducing additional levels of tailoring complexity by gray or even glass-boxing atomic components. Thus, selected aspects or even the whole code of an atomic component could become modifiable by certain users. With regard to the user interface for tailoring, one has to investigate whether a single interaction paradigm is sufficient for component-based tailorability, or whether the interaction paradigm of the tailorable application needs to be taken into account for the design of the interface of the tailoring environment (cf. Nardi, 1993).

Author's personal copy

At the interface level, we aimed to increase the transparency regarding technology structure and possible technology modifications (gray boxing, component connectors), and regarding the effects of modifications (by means of Exploration Environments and visualizations of certain additional component dependencies). A dynamic well known in the field of CSCW is that additional transparency always favors a loss of privacy, e.g. the more accurate an Exploration Environment reflects the actual state of a groupware application within an organization, the more information about ongoing usages and even about users may become available. The perceived value of user interface concepts we suggested will not only depend on the adding of transparency, but also on controlling transparency.

Regarding the support of collaborative tailoring activities, we need to think of additional features that support users in selecting appropriate atomic or abstract components out of a larger set of components. Ye (2001) has developed a recommender system which supports software developers to share and reuse source codes via a repository. We believe that similar functionalities will be valuable for collaborative tailoring activities. Moreover, we will be able to learn from the open source movement and the discussion on social capital to design shared repositories in an appropriate manner (cf. Fischer et al., 2004; Huysman and Wulf, 2004).

6.3. *Beyond component-based systems*

The final point we want to make concerns software development paradigms. One important issue is the development of appropriate tailoring platforms for peer-to-peer architectures and mobile systems (cf. Alda and Cremers, 2004). Our approach may fall short since we use a client-server architecture. However, peer-to-peer architectures face additional challenges regarding synchronization and availability.

We started with a set of ideas and requirements that originated in CSCW research and combined them with issues of software reuse in software engineering. It leads us to new requirements for component-based systems that we addressed with the FlexiBeans wrappers, the FreEvolve platform, a set of 'congruencies' guiding the user interface design and additional ideas for collaboration support. The great effort we had to make to produce acceptable solutions for our requirements is due to the fact that the necessity to design for 'redesign during use' is not an important consideration in software engineering. Several problems we addressed could be more substantially resolved by integrating end user concerns in the development of standards and programming languages. With the emergence of software-oriented architectures (SOA), new promises and challenges are bound to occur. Our experiences have shown how difficult it was to provide an infrastructure that addresses the two main goals of end user orientation and run-time tailorability within component networks. SOA promise a higher level of flexibility since the concept is not related to an operating system or programming language (as component-based systems are). However, the current practice of SOA restricts itself to a use by programmers rather than by end users. There was a similar situation when we started our work with components. To some extent, SOA resemble aspects of component-based architectures: services are independent of each other, stateless, self-describing and provide standardized interfaces (Brown et al., 2002; Bieberstein et al., 2006). However, services provide additionally a process perspective on applications, not just an object or component perspective (Jones and Mike, 2005). The promise of a new level of flexibility is already partly being undermined by an inflationary use of the term 'service-orientation' (Doernhoefer, 2006). Standardization efforts are counteracted in practice by the emergence of different service worlds (e.g. SOAP web services vs. OSGi architecture). It is an interesting research issue to investigate how the concepts developed for component-based systems translate into the SOA world.

Taking into account the emergence of SOA as a new flexibilization technology that again needs to be bended to fit the user needs, the question arises whether it is an appropriate research strategy to 'abuse' technologies that have been developed to address the issue of software reuse. As a result of our work, the research challenges of use-time redesign by the end user became obvious. It seems to be possible to make modern flexibilization technologies better manageable by end users.

However, it is not only the technological implementations that need to be improved in order to make things easier for 'casual programmers'. The implementations for software reuse in professional software development need to focus on different dynamics (e.g. interoperability, backward compatibility). Moreover, even simpler software applications rely on underlying layers of software (e.g. operating systems, interface frameworks, communication protocols) and result in a system in which it is inherently difficult to navigate, interpret its visible effects and manipulate applications. If software technology is that complex, the question needs to be asked whether we can really bend flexibilization technologies such as SOA to a level at which end users can reliably act with them. To deal with these challenges, we believe that beyond technology user communities need to develop around software applications (cf. Stevens and Wiedenhöfer, 2006). We may need to fundamentally shift our usage traditions of information technology and establish 'tailoring' and 'programming' to become cultural techniques just as reading or counting. If technology cannot become more intelligible, users (communities) have to develop their abilities. Since trends such as ubiquitous computing increase IT diffusion into almost all aspects of our lives, tailoring as well as programming skills should be developed as early as in school.

7. Conclusions

We have presented our work on how the concept of component-based tailorability can be made intelligible and

Author's personal copy

manageable for end users. Due to the specific requirements of users whose main interests are other than software development, the requirements for the design of the user interface are distinct from typical developer-oriented IDEs in software engineering. We worked out a component-based approach by evaluating our experiences with the FreeEvolve platform. In addition, we developed a number of prototypes that covered important side issues. These issues covered constraint-based integrity checks in order to help users to detect tailoring errors, as well as the provision of Exploration Environments, so they can familiarize themselves with the dynamic aspects of tailored component networks.

We suggested a holistic approach to component-based tailorability by addressing the architectural level as well as the interface and the social/collaborative level. We described how we used different 'congruencies' (Component-Metaphor Congruency, Architecture-Interface Congruency and Tailoring-Use Congruency) as guidelines to provide a comprehensive tailoring environment complemented with the interface concept of direct activation. We discussed the role of these congruencies in the development of three different tailoring interfaces. Furthermore, we addressed the social level by providing shared repositories of tailored components, and connecting them to Exploration Environments.

Finally, we related our findings to the issue of developing technology in an 'infrastructure-aware' way. It is necessary to consider technology as an infrastructure for users, as something that remains invisible until breakdown, in which case it becomes critical to know. Such a perspective promotes several notions for the support of 'appropriation work', which users perform typically when making sense of technologies. Perceiving groupware as an infrastructure also means that the levels of qualification, interest, and dedication will be different among users involved in tailoring an application. They will also vary over time. The infrastructure issue connects to earlier discussions about the need of an emerging tailoring culture within the field of application (cf. Carter and Henderson, 1990).

Along these lines of thought, we need to gather new experiences on how to connect tailoring activities with processes of organizational development and change (cf. Wulf and Jarke, 2004). In order to improve flexibility and efficiency of business processes, the exploitation of tailorability needs to be integrated into the ongoing processes of organizational changes. Therefore, we have developed the framework of Integrated Organization and Technology Development which connects tailorability with planned processes of organizational and technological development (cf. Wulf and Rohde, 1995; Rohde, 2007). This aspect needs to be further investigated particularly regarding emerging change processes and the role of tailorability in groupware appropriation (cf. Orlikowski and Hofman, 1997; Andriessen et al., 2003; Dittrich et al., 2005; Pipek et al., 2006).

Our research demonstrates that there is a quality of software, offering new opportunities for the integration of the traditionally separated spheres of 'design' and 'use' that go far beyond the capabilities of products with a 'real' materiality. Software can be considered a new material which carries with it the ability to change beyond the ideas and intentions of the original designers. This material can incorporate communication and collaboration channels, which support its development and appropriation. Its interfaces may melt well into existing technological and social infrastructures. With its ubiquity, this material and its products may well become the 'boundary objects' for reconceptualizations of reality that reach beyond the technological level. This notion opens up new horizons for capturing and managing innovations, and for mediating developments on an organizational as well as societal level. These horizons are yet to be explored.

Acknowledgements

Torsten Engelskirchen, Björn Golombek, Michael Hallenberger, Ralf Hinken, Jörg-Guido Höpfner, Helge Kahler, Markus Rohde, Gunnar Stevens, and Oliver Stiemerling contributed greatly to different aspects of the work presented here. The authors gratefully acknowledge the support of the German Science Foundation (DFG) and the German Ministry of Research and Education (BMBF). BMBF has provided grants in the context of the 'PoliTeam: Workflows, Archives and Groupwork in Distributed Organizations' and the 'EUDISMES: End User Development in Small and Medium Enterprise Software Systems' research projects. DFG funds the research projects 'The Design of Computer Systems by the User' as part of the DFG Research Centre 'Media Upheavals' (DFG-FK 615) and the project cluster 'Context-Aware Interaction in Cooperative Knowledge Processes (Contici)'. The authors would also like to thank the journal editor and three anonymous reviewers for their constructive suggestions, as well as Fred Kop for his detailed editorial comments.

References

Ackermann, D., Ulich, E., 1987. The chances of individualization in human–computer interaction and its consequences. In: Frese, M., Ulich, E., Dzida, W. (Eds.), Psychological Issues of Human Computer Interaction in the Work Place. North-Holland, Amsterdam, pp. 131–146.

Alda, S., Cremers, A.B., 2004. Towards composition management for peer-to-peer architectures. In: Proceedings of the Workshop Software Composition (SC 2004), affiliated to the Seventh European Joint Conference on Theory and Practice of Software (ETAPS 2004), Barcelona, Spain.

Ambler, A., Leopold, J., 1998. Public Programming in a Web World. Visual Languages. Nova Scotia, Canada.

Andriessen, J.H.E., Hettinga, M., Wulf, V. (Eds.), 2003. Special issue on evolving use of groupware. Computer Supported Cooperative Work: The Journal of Collaborative Computing (JCSCW), 12(4).

Bellissard, L., Atallah, S.B., Boyer, F., Riveill, M., 1996. Distributed application configuration. Proceedings of the 16th International Conference on Distributed Computing Systems. IEEE-Press, Hongkonk, pp. 579–585.

Author's personal copy

Bentley, R., Dourish, P., 1995. Medium versus mechanism. supporting collaboration through customisation. In: Marmolin, H., Sundblad, Y., Schmidt, K. (Eds.), Proceedings of the Fourth European Conference on Computer Supported Cooperative Work—ECSCW '95, Kluwer, pp. 133–148.

Bentley, R., Appelt, W., Busbach, U., Hinrichs, E., Kerr, D., Sikkel, K., Trevor, J., Woetzel, G., 1997. Basic support for cooperative work on the world wide web. International Journal of Human Computer Studies 46, 827–846.

Beringer, J., 2004. Reducing expertise tension. Communications of the ACM 47 (9), 39–40.

Bieberstein, N., Bose, S., Fiammante, M., Jones, K., Shah, R., 2006. Service-Oriented Architecture Compass. Pearson.

Blomberg, J., Giacomi, J., Mosher, A., Swenton-Wall, P., 1993. Ethnographic field methods and their relation to design. In: Schuler, D., Namioka, A. (Eds.), Participatory Design: Principles and Practices. Lawrence Earlbaum Assoc., Hillsdale, NJ, pp. 123–156.

Brown, A.W., Johnston, S., Kelly, K., 2002. Large-scale, using service-oriented architecture and component-based development to build web service applications. Rational Software White Paper TP032.

Burton, R.R., Brown, J.S., Fischer, G., 1984. Skiing as a model of instruction. In: Rogoff, B., Lave, J. (Eds.), Everyday Cognition: Its Development in Social Context. Harvard University Press, Cambridge, MA, pp. 139–150.

Carroll, J.M., 1987. Five gambits for the advisory interfaces dilemma. In: Frese, M., Ulich, E., Dzida, W. (Eds.), Psychological Issues of Human Computer Interaction in the Work Place, Amsterdam, pp. 257–274.

Carroll, J.M., Carrithers, C., 1984. Training wheels in a user interface. Communications of the ACM 27 (8), 800–806.

Carter, K., Henderson, A., 1990. Tailoring culture. In: Hellman, R., Ruohonen, M., Sorgard, P. (Eds.), Proceedings of the 13th IRIS, Reports on Computer Science and Mathematics, No. 107, Abo Akademi University, pp. 103–116.

Cortes, M., 1999. A coordination language for building collaborative applications. Journal of Computer Supported Cooperative Work.

Dittrich, Y., Dourish, P., Mørch, A., Pipek, V., Stevens, G., Törpel, B., 2005. Special issue on supporting appropriation work. International Reports on Socio-Informatics (IRSI) (2:2), p. 84, 〈http://irsi.iisi.de/〉.

Doernhoefer, M., 2006. Surfing the net for software engineering notes. ACM SIGSOFT Software Engineering Notes, vol. 30, issue 6, pp. 5–13, November 2005. ISSN:0163-5948.

Dourish, P., 1996. Open implementation and flexibility in CSCW toolkits. Ph.D. Thesis, University College, London.

Engelskirchen, T., 2000. Exploration anpassbarer groupware. Master Thesis, University of Bonn.

Fischer, G., Girgensohn, A., 1990. End-user modifiability in design environments. In: Proceedings of the Conference on Computer Human Interaction (CHI ′90), 1–5 April 1990, Seattle, Washington. ACM-Press, New York, pp. 183–191.

Fischer, G., Lemke, A.C., Rathke, C., 1987. From design to redesign. In: International Conference on Software Engineering, Monterey, CA, USA, pp. 369–376.

Fischer, G., Scharff, E., Ye, Y., 2004. Fostering social creativity by increasing social capital. In: Huysman, M., Wulf, V. (Eds.), Social Capital and Information Technology. MIT-Press, Cambridge, MA, pp. 355–399.

Frese, M., Irmer, C., Prümper, J., 1991. Das Konzept Fehlermanagement: Eine Strategie des Umgangs mit Handlungsfehlern in der Mensch-Computer Interaktion ("The concept of ,,Fault Management": A strategy of dealing with activity faults in Human–Computer-Interactions'). In: Skarpelis, C. (Ed.), Software für die Arbeit von morgen (Software for tomorrow's work'). Springer, Berlin, pp. 241–252.

Golombek, B., 2000. Implementierung und Evaluation der Konzepte "Explorative Ausführbarkeit" und "Direkte Aktivierbarkeit" für anpassbare Groupware ('Implementation and evaluation of the concepts of "explorative execution" and "direct activation" for tailorable groupware'). Master Thesis, University of Bonn.

Grudin, J., 1996. Evaluating Opportunities for Design Capture. In: Moran, J.P., Carroll, J.M. (Eds.), Design Rationale: Concepts, Techniques and Use, p. Hillsdale.

Hallenberger, M., 2000. Programmierung einer interaktiven 3D-Schnittstelle am Beispiel einer Anpassungsschnittstelle für komponentenbasierte Anpassbarkeit ('Implementation of an interactive 3D-interface in the example of a tailoring interface for component-based tailorability'). Master Thesis, University of Bonn.

Henderson, A., Kyng, M., 1991. There's no place like home: continuing design in use. In: Greenbaum, J., Kyng, M. (Eds.), Design At Work—Cooperative Design of Computer Artefacts. Lawrence Erlbaum Associates, Hillsdale, NJ, pp. 219–240.

Hinken, R., 1999. Verteilte Anpassbarkeit für Groupware—Eine Laufzeit und Anpassungsplattform ('Distributed tailoring of groupware—a run-time and tailoring environment'). Master Thesis, University of Bonn.

Höpfner, J.-G., 1998. Gemeinsame Anpassung von Einzelpatzanwendungen ("Collaborative tailorability of single user applications"). Master Thesis, University of Bonn.

Howes, A., Paynes, S.J., 1990. Supporting exploratory learning. In: Proceedings of INTERACT'90, North-Holland, Amsterdam 1990, pp. 881–885.

Huysman, M., Wulf, V. (Eds.), 2004. Social Capital and Information Technology. MIT Press, Cambridge, MA.

IBM, 1998. Visual Age for Java, Version 1.0.

ISO, 9241, 1999. Ergonomic Requirements for Office Work with Visual Display Terminals (VDTs), Part 10: Dialogue Principles.

Johansen, R., 1988. Current user approaches to groupware. In: Johansen, R. (Ed.), Groupware. Freepress, New York, pp. 12–44.

Jones, S., Mike, M., 2005. A methodology for service architectures. OASIS Draft, 〈http://www.oasis-open.org/committees/download.php/15071/A%20methodology%20for%20Service%20Architectures%201%202%204%20-%20OASIS%20Contribution.pdf〉, referenced in January 2006.

Kahler, H., 2000. Methods and tools: constructive interaction and collaborative work: introducing a method for testing collaborative systems. Interactions 7 (3), 27–34.

Kahler, H., 2001a. Supporting collaborative tailoring. Ph.D. Thesis, Roskilde University, Denmark, Roskilde.

Kahler, H., 2001b. More than WORDSs: collaborative tailoring of a word processor. Journal on Universal Computer Science (j.ucs) 7 (9), 826–847.

Krings, M., 2002. Erkennung semantischer Fehler in komponentenbasierten Architekturen ('The recognition of semantic errors in component-based architectures'). Master Thesis, University of Bonn.

Krüger, M., 2003. Semantische Integritätsprüfung für die Anpassung von Komponenten-Kompositionen ('Semantic integrity checking for tailoring component aggregates'). Master Thesis, University of Bonn.

Kuutti, K., Bannon, L., 1993. Searching for Unity Among Diversity: Exploring the Interface Concept, interchi'93. ACM Press, pp. 263–268.

Lewis, C., 1982. Using the "Thinking-aloud" method in cognitive interface design. Research Report RC 9265. T.J. Watson Research Center, Yorktown Heights, NY.

Lieberman, H., Paternó, F., Wulf, V. (Eds.) 2006. End User Development. Springer, London.

Mackay, W.E., 1990. Users and customizable software: a co-adaptive phenomenon. Ph.D. Thesis, MIT, Boston, MA.

MacLean, A., Carter, K., Lövstrand, L., Moran, T., 1990. User-tailorable systems: Pressing the issue with buttons. In: Proceedings of the Conference on Computer Human Interaction (CHI '90), 1–5 April, Seattle (Washington). ACM Press, New York, pp. 175–182.

Magee, J., Dulay, N., Eisenbach, S., Kramer, J., 1995. Specifying distributed software architectures. In: Proceedings of Fifth European Software Engineering Conference, Barcelona.

Malone, T.W., Lai, K.-Y., Fry, C., 1992. Experiments with oval: a radically tailorable tool for cooperative work. In: Proceedings of CSCW, Toronto, Canada. ACM Press, pp. 289–297.

Author's personal copy

McIlroy, D., 1968. Mass-produced software components. In: Proceedings of Conference on Software Engineering, Garmisch-Partenkirchen (D), North Atlantic Treaty Organization (NATO).

Microsoft, 1996. Visual Basic, Version 4.0.

Mørch, A.I., 1997. Method and tools for tailoring of object-oriented applications: an evolving artifacts approach. Ph.D. Thesis, Department of Computer Science, University of Oslo, Research Report 241, Oslo.

Mørch, A.I., Mehandjiev, N.D., 2000. Tailoring as collaboration: the mediating role of multiple representations and application units. Computer Supported Cooperative Work 9 (1), 75–100.

Mørch, A.I., Stevens, G., Won, M., Klann, M., Dittrich, Y., Wulf, V., 2004. Component-based technologies for end user development. Communications of the ACM 47 (9), 59–62.

Myers, B.A., 1990. Taxonomies of Visual Programming and program visualization. Journal of Visual Languages and Computing 1, 97–123.

Nardi, B.A., 1993. A Small Matter of Programming—Perspectives on End-User Computing. MIT Press, Cambridge.

Nielsen, J., 1993. Usability Engineering. Academic Press, Boston, MA.

Oberquelle, H., 1993. Anpassbarkeit von Groupware als Basis für die dynamische Gestaltung von computergestützter Gruppenarbeit ('Tailorability of Groupware as a basis for a dynamic design of Computer-Supported Cooperative Work'). In: Konradt, U., Drisis, L. (Eds.), Benutzeroberflächen in der teilautonomen Arbeit (User Interfaces in Semi-Autonomous Work). Köln, pp. 37–54.

Oberquelle, H., 1994. Situationsbedingte und benutzerorientierte Anpassbarkeit von Groupware (Situational and user-oriented tailorability of Groupware), In: Hartmann, A., Herrmann, Th., Rohde, M., Wulf, V. (Eds.), Menschengerechte Groupware. Stuttgart, pp. 31–50.

Oppermann, R., Simm, H., 1994. Adaptability: user-initiated individualization. In: Oppermann, R. (Ed.), Adaptive User Support—Ergonomic Design of Manually and Automatically Adaptable Software. LEA, Hillsdale, NJ.

Orlikowski, W.J., Hofman, J.D., 1997. An improvisational model for change management: the case of groupware technologies. Sloan Management Review (Winter 1997), 11–21.

Page, S., Johnsgard, T., Albert, U., Allen, C., 1996. User customization of a Word Processor. In: Proceedings of CHI '96, 13–18 April, pp. 340–346.

Pane, J.F., Myers, B.A., 2006. More natural programming languages and environments. In: Lieberman, H., Paternó, F., Wulf, V. (Eds.), End User Development. Kluwer, Dordrecht, pp. 31–50.

Pane, J.F., Myers, B.A., Ratanamahatana, C.A., 2001. Studying the language and structure in non-programmers' solutions to programming problems. International Journal of Human–Computer Studies 54 (2), 237–264.

Paul, H., 1994. Exploratives Agieren ("Explorative Agency"). Peter Lang, Frankfurt.

Pipek, V., 2003. An integrated design environment for collaborative tailoring. In: Dosch, W., Lee, R.Y. (Eds.), ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD'03). ACIS, Lübeck, Germany, pp. 430–438.

Pipek, V., 2005. From tailoring to appropriation support: negotiating groupware usage. In: Faculty of Science, Department of Information Processing Science (ACTA UNIVERSITATIS OULUENSIS A 430), University of Oulu, Oulu, Finland.

Pipek, V., Kahler, H., 2006. Supporting collaborative tailoring. In: Lieberman, H., Paternó, F., Wulf, V. (Eds.), End User Development. Kluwer, Dordrecht.

Pipek, V., Syrjänen, A.-L., 2006. Infrastructuring as capturing in-situ design. In: Proceedings of 7th Mediterranean Conference on Information Systems. Association of Information Systems, Venice, Italy. Online at 〈http://www.aisworld.org/〉.

Pipek, V., Rosson, M.B., Stevens, G., Wulf, V., 2006. Supporting the appropriation of ICT—end-user development in civil societies. Journal of Community Informatics (2:2).

Ravichandran, T., Rothenberger, M.A., 2003. Software reuse strategies and component markets. Communications of the ACM 46 (8), 109–114.

Repenning, A., Ioannidou, A., Zola, J., 2000. AgentSheets: end-user programmable simulations. Journal of Artificial Societies and Social Simulation 3 (3).

Robertson, T., 1998. Shoppers and tailors: participative practices in small Australian design companies. Computer Supported Cooperative Work (CSCW) 7 (3–4), 205–221.

Rohde, M., 2007. Integrated Organization and Technology Development (OTD) and the Impact of Socio-Cultural Concepts—A CSCW Perspective, Datalogiske skrifter. University of Roskilde, Roskilde, Denmark.

Schmidt, K., 1991. Riding a Tiger or Computer Supported Cooperative Work. In: Bannon, L., Robinson, M., Schmidt, K. (Eds.), Proceedings ECSCW '91, Kluwer, Dordrecht, pp. 1–16.

Shu, N.C., 1988. Visual Programming. Van Nostrand Reinhold Co., New York, NY.

Silberschatz, A., Korth, H., Sudarshan, S., 2001. Database System Concepts. McGraw-Hill, Osborne.

Slagter, R., Biemans, M., Ter Hofte, G.H., 2001. Evolution in use of groupware: facilitating tailoring to the extreme. In: Borges, M., Haake, J., Hoppe, U. (Eds.), Proceedings of the Seventh International Workshop on Groupware (CRIWG 2001), 6–8 September 2001, Darmstadt, Germany.

Star, S.L., Bowker, G.C., 2002. How to infrastructure. In: Lievrouw, L.A., Livingstone, S. (Eds.), Handbook of New Media—Social Shaping and Consequences of ICTs. Sage Publication, London, UK, pp. 151–162.

Steven, G., 2002. Komponentenbasierte Anpassbarkeit—FlexiBeans zur Realisierung einer erweiterten Zugriffskontrolle ('Component-based tailorability—using flexibeans to implement an extended access control system'). Master Thesis, University of Bonn.

Stevens, G., Wiedenhöfer, T., 2006. CHIC—a pluggable solution for community help in context. In: Mørch, A.I., Morgan, K., Bratteteig, T., Ghosh, G., Svanaes, D. (Eds.), Proceedings of the Fourth Nordic Conference on Human-Computer Interaction (NordiCHI 2006), Oslo, Norway, 14–18 October 2006, pp. 212–221.

Stevens, G., Wulf, V., 2002. A new dimension in access control: studying maintenance engineering across organizational boundaries. In: Proceedings of ACM Conference on Computer Supported Cooperative Work (CSCW 2002). ACM Press, New York, pp. 196–205.

Stevens, G., Quaisser, G., Klann, M., 2006. Modulizing software for tailorability—an industrial case study. In: Lieberman, H., Paternó, F., Wulf, V. (Eds.), End User Development. Kluwer, Dordrecht, pp. 269–294.

Stiemerling, O., 1997. CAT: component architecture for tailorability. Working Paper, Department of Computer Science, University of Bonn.

Stiemerling, O., 1998. FlexiBeans specification V 2.0. Working Paper, Department of Computer Science, University of Bonn.

Stiemerling, O., 2000. Component-based tailorability. Ph.D. Thesis, Department of Computer Science, University of Bonn, Bonn. Available from: 〈http://www.freevolve.de/Dissertation.pdf〉.

Stiemerling, O., Cremers, A.B., 1998. Tailorable component architectures for CSCW-systems. In: Tyrell, A.M. (Ed.), Proceedings of the Sixth Euromicro Workshop on Parallel and Distributed Processing. IEEE-Press, pp. 302–308.

Stiemerling, O., Hinken, R., Cremers, A.B., 1999a. The Evolve tailoring platform: supporting the evolution of component-based groupware. In: Proceedings of EDOC'99. IEEE Press, Mannheim, 27–30 September 1999, pp. 106–115.

Stiemerling, O., Hinken, R., Cremers, A.B., 1999b. Distributed component-based tailorability of CSCW applications. In: Proceedings of ISDS'99, Tokio, Japan. IEEE-Press, pp. 345–352.

Stiemerling, O., Kahler, H., Wulf, V., 1997. How to make software softer—designing tailorable applications. In Proceedings of Second Conference on the Design of Interactive Systems, Amsterdam (NL). ACM Press, pp. 365–376.

Stiemerling, O., Won, M., Wulf, V., 2000. Zugriffskontrolle in Groupware—Ein nutzerorientierter Ansatz (Access control in Groupware—a

Author's personal copy

user-centered approach). In: WIRTSCHAFTSINFORMATIK, 42. Jg., Nr. 4, 2000, pp. 318–328.

Stiemerling, O., Hallenberger, M., Cremers A.B., 2001. 3D interface for the administration of component-bases, distributed systems. In: Proceedings of Fifth International Symposium on Autonomous Decentralized Systems, 26–28 March 2001, Dallas, TX. IEEE Press, pp. 119–126.

Szyperski, C., 2002. Component Software: Beyond Object Oriented Programming, second ed. Addison-Wesley, London.

Teege, G., 2000. Users as composers: parts and features as a basis for tailorability in CSCW systems, CSCW. Kluwer Academic Publishers, pp. 101–122.

van der Aalst, W.D.H., Verbeek, H.M.W., 1997. A Petri-Net-based tool to analyze workflows. In: Proceedings of Petri Nets in System Engineering (PNSE'97), Universität Hamburg, Hamburg, 1997, pp. 78–90.

Wang, W., Haake, J.M., 2000. Tailoring Groupware: The Cooperative Hypermedia Approach. International Journal of Computer-Supported Cooperative Work (9:1).

Won, M., 1998. Komponentenbasierte Anpassbarkeit—Anwendung auf ein Suchtool für Groupware ('Component-based tailorability and Ist application on a groupware searching tool'). Master Thesis, University of Bonn.

Won, M., 2000. Checking integrity of component-based architectures. CSCW 2000 Philadelphia, USA, Workshop on Component-Based Groupware.

Won, M., 2003. Supporting end-user development of component-based software by checking semantic integrity. In: ASERC Workshop on Software Testing, 19.2.2003, Banff, Canada.

Won, M., Stiemerling, O., Wulf, V., 2006. Component-based approaches to tailorable systems. In: Lieberman, H., Paternó, F., Wulf, V. (Eds.), End User Development. Springer, London, pp. 127–153.

Wulf, V., 1994. Anpaßbarkeit im Prozeß evolutionärer Systementwicklung ('Tailorability Within the Process of Evolutionary System Development'). GMD-Spiegel, vol. 24, 3/94, pp. 41–46.

Wulf, V., 1999. Let's see your Search-Tool!—collaborative use of tailored artifacts in groupware. In: Proceedings of GROUP '99. ACM Press, New York, 1999, pp. 50–60.

Wulf, V., 2000. Exploration environments: supporting users to learn groupware functions. Interacting with Computers 13 (2), 265–299.

Wulf, V., 2001. Zur anpassbaren Gestaltung von Groupware: Anforderungen, Konzepte, Implementierungen und Evaluationen ('On the design of tailorable Groupware: Requirements, Concepts, Implementations and Evaluations'), GMD Research Series, Nr. 10/2001, St. Augustin, 2001 (and: Habilitation Thesis University of Hamburg 2000).

Wulf, V., Golombek, B., 2001a. Direct activation: a concept to encourage tailoring activities. Behavior and Information Technology 20 (4), 249–263.

Wulf, V., Golombek, B., 2001b. Exploration environments—concept and empirical evaluation. In: Proceedings of GROUP 2001. ACM Press, New York, pp. 107–116.

Wulf, V., Jarke, M., 2004. The economics of end-user development. Communications of the ACM 47 (9), 41–42.

Wulf, V., Rohde, M., 1995. Towards an integrated organization and technology development. In: Proceedings of the Symposium on Designing Interactive Systems, 23–25 October 1995, Ann Arbor (Michigan). ACM Press, New York, pp. 55–64.

Wulf, V., Stiemerling, O., Pfeifer, A., 1999. Tailoring groupware for different scopes of validity. Behaviour and Information Technology 18 (3), 199–212.

Wulf, V., Kahler, H., Stiemerling, O., Won, M., 2005. Tailoring by integration of domain-specific components: the case of a document search tool. Behaviour and Information Technology (BIT) 24 (4), 317–333.

Yang, Y., 1990. Current approaches and new guidelines for undo-support design. In: Proceedings of INTERACT'90. North-Holland, Amsterdam, pp. 543–548.

Ye, Y., 2001. Supporting component-based software development with active component repository systems. Ph.D. Dissertation, Department of Computer Science, University of Colorado at Boulder, Boulder.